GLENTOP

SPECTRUM 128 Companion

Ian Sinclair

INCLUDING 128 PLUS-2

THE SPECTRUM 128 COMPANION

THE SPECTRUM 128 COMPANION

by

Ian Sinclair

Glentop Publishers Ltd

MAY 1986

ISBN 1 85181 080 3

Published by:
Glentop Publishers Ltd
Standfast House
Bath Place
High Street
Barnet
Herts EN5 5XE
Tel: 01–441–4130

Printed and bound in Great Britain by
Richard Clay Ltd, Bungay, Suffolk

Contents

Preface

The Spectrum 128 is the ultimate Spectrum machine. It combines the simplicity of the early Spectrum with the better keyboard of the Spectrum Plus, and adds a sound system which, for my money, is the most satisfactory that I have ever used. In addition, the memory size has been doubled. This does not allow you to write longer programs in BASIC, but it allows very long machine-code programs to be used for games and other purposes, and also permits you to keep several BASIC programs in use at one time, switching from one to another as you want.

All in all, the differences between the older Spectrum and the new 128 model are such that even the owner of a Spectrum Plus will find much that needs to be re-learned on this new machine. Those of you who are approaching the Spectrum 128 as a first computer will, of course, find that this is all new. The early Spectrum had a large and very comprehensive Manual, but as the years have gone by this has shrunk considerably, and the latest version is mainly aimed at the user who never programs. I shall assume that you want to get the best value for your money, so you'll want to program the machine. For that purpose, you'll want to use it in its 128 Mode. The Spectrum 128 allows you to use it, in its 48 BASIC mode, as if it were a Spectrum Plus. This makes it easy to use by the former owner of a Spectrum Plus, but does not allow you to use the advanced features that are possible only when you switch to the 128 mode.

One very welcome feature of the 128 mode is that it allows the keyboard to be used in the way that all other computers use it – to type out words in full. You no longer need to remember odd combinations of keys, and by working with your Spectrum 128 you will be getting good practice at the skills that you will need to use other larger machines, such as the bigger Amstrad models, and the new generation of machines that are appearing in Schools and Colleges. In the Spectrum 128, then, you have a very able computer that will be an excellent preparation for using any other machines, and which will prove to be very useful in its own right. The only snag at present is the use of tape cassettes, but I am sure that this will eventually be remedied.

Finally, I am very grateful to a number of people without whose assistance this book would not have been produced. To Andy Savery and Peter Holmes at Glentop Publishers, I am most grateful for support, encouragement, the commissioning of this book – and the loan of a printer cable. I am also considerably obliged to Mark Newton at Sudbury Microsystems, who produced a Spectrum 128 for me at a moment's notice. I must also record my gratitude to Alan Giles, of Melbourne House, who knew the commands to get my printer working, and so saved me many hours of work.

Ian Sinclair
Spring 1986

Chapter 1

Setting up the Spectrum 128

Your Spectrum 128 computer comes exceptionally well packed, and the package (at the time of writing) contains the Spectrum 128 itself (from now on, we'll just call it the Spectrum), its power supply and the TV cable, together with a cassette-recorder cable, two slim manuals, and two sample games. It's possible that later versions may include other accessories, because there is space in the packaging for other items. The manual contains instructions for helping the beginner to run games tapes, but not a lot on programming the machine or on setting up TVs and cassette recorders to get the best results. Unless you have a lot of experience with computers, you will probably need some more advice than the manual (which is very much better than most computer manuals) can spare space for.

The power supply is the large black cube with the two sets of cables. One of these cables is connected permanently to a small plug. This is the plug that fits into the Spectrum keyboard at the rear right-hand side (looking from the front). The socket for this plug is labelled '9VDC', and the plug should be inserted carefully and firmly all the way in. Don't on any account use excessive force; the plug should be a firm (though not tight) fit. If you apply a lot of force, it is possible to damage the socket inside the computer. The power supply cube is quite heavy, and you should be careful not to lift the Spectrum without lifting or disconnecting the power supply as well. The other cable from the power supply must be fitted with a standard three-pin mains plug before you can start to make use of your Spectrum. Once you have attached the power supply cable to your computer, you should try to keep it in place. You can switch the whole computer on and off by a switch on the mains socket if there is one, or by plugging or unplugging the mains plug.

The plug is connected as indicated in Figure 1.1. There are only two leads, one blue and the other brown, and the cable should be tightly clamped. The fuse should be a 3 amp type, not the 13 amp variety that usually comes with the three-pin plug. If you are accustomed to fitting plugs for yourself then the diagram should be enough to remind you of what is needed. If you don't want to have anything to do with mains supplies, then take the Power Supply Unit along to an electrician and get a plug, with a 3A fuse, connected. You don't have to take the whole computer, only the power

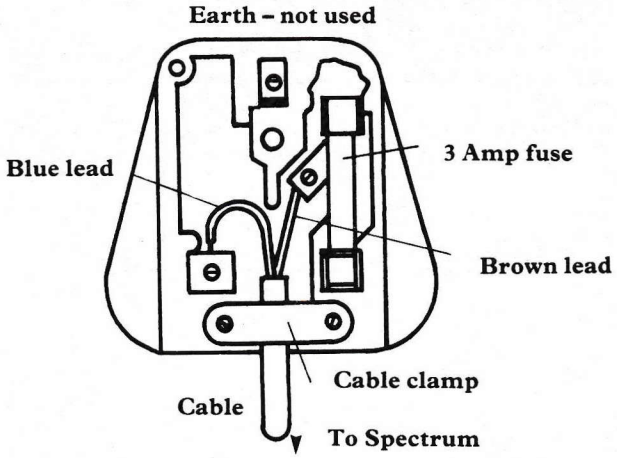


Figure 1.1 The connections to a three-pin mains plug. Only the live and neutral leads are used. If you haven't done this sort of thing before, play safe and hand it to an electrician.

supply box. When you have done this, make sure that the power supply box is located away from the Spectrum 128 and clear of the TV. Put it where the air can circulate round it. It gets only slightly warm while you are using the computer, but you should always try to keep it in a cool place. You may also want to place it on a felt pad or a polystyrene block, because if it rests on a table, it gives out an irritating humming or buzzing noise. You should also locate the computer itself so that the fins on the right-hand side are clear of any obstacles. These are cooling fins, and it should be possible for the air to circulate around them freely.

With that hurdle over, you are almost ready to put the Spectrum 128 to work for you, but you need the use of a TV receiver or a monitor. A computer is a device which is arranged so as to send out electrical signals that can be used to form images on a TV screen. There are two ways in which this can be done. One is to use a special form of TV display, which is designed to use the signals from the computer directly. Such a device is called a *monitor*, and it can't normally be used for receiving TV pictures from an aerial. The alternative method is to convert the signals from the computer into the same form as the signals that TV transmitters send, so that they can be connected to the aerial socket of an ordinary TV receiver.

There is an important difference between the two. Though it's convenient to be able to use a TV receiver (and cheap as well), the picture that you see is of a poor standard. This is because a TV receiver was never designed to accept computer signals, and also because of the need to convert the computer's signals into the same form as transmitted signals. The result is that the letters and other shapes on the screen look fuzzy, and the colours look streaky. If you use a monitor, connected to the Spectrum 128 with a suitable cable, the shapes on the screen will be clearer, and the colours much better. This is a connection which your local computer store will have to help you with, because the Sinclair monitor cable (an extra) comes with only one plug, the one at the Spectrum 128 end. The computer store will have to fit the other plug which fits into the monitor. At the time of writing this book, I could not obtain suitable connectors, and I used a Fidelity MTV100, which is a combined TV and monitor, but with the TV input only.

Seeing is believing

Unless you connect a TV receiver or monitor to the Spectrum you won't be able to see what the computer is doing. It will still compute for you just as well, but you won't see what is going on. To connect the Spectrum to a TV receiver, you will need to plug the aerial lead to both the Spectrum and the TV. Unless you can keep a TV receiver specially for use with the Spectrum, you will find that you have to keep plugging and unplugging the aerial cable and the Spectrum cable. This is never a good thing to have to do, because it wears out the socket on the TV, and a useful alternative is to use the type of adaptor that is illustrated in Figure 1.2. This allows you to plug a lead into the aerial socket of the TV so that the TV can be used both for Spectrum and for *Dallas* without having to pull plugs out. The two-way TV adaptor that I used is sold in TV stores under the name of *PANDA*. You need to connect the Spectrum to the TV or to the adaptor using the special cable that is provided with the Spectrum. There are two different plugs on this cable and Figure 1.3 shows the difference between these plugs. If you have used the adaptor, then all that you have to do to change between computing and TV watching is to change channels!

The TV or monitor that you use to display the Spectrum's signals need not be a colour receiver, not to start with at least. The skills of programming a Spectrum do not require you to see the results in colour until you come to the colour instructions of the Spectrum in Chapter 8. When you use a black-and-white TV or monitor to show the Spectrum signals, the colours appear as shades of grey, and they are quite distinct. If you use a colour receiver, you will see the colours appear in all their glory, though not all makes of TV receivers will give equally good displays.

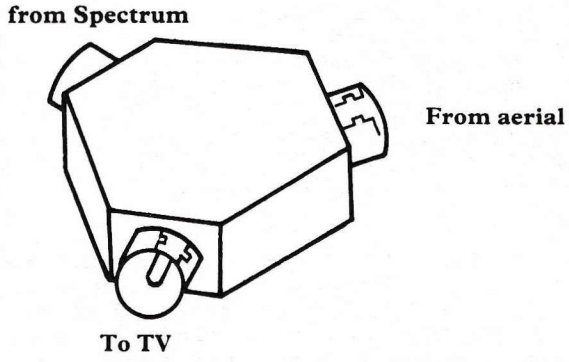


Figure 1.2 A TV aerial cable adaptor, which allows you to keep both the computer and the aerial connected.

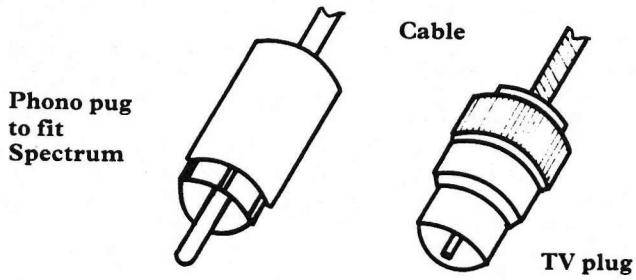


Figure 1.3 The different plugs at the two ends of the TV connecting cable.

The big switch-on

Now before you plug in everything in sight and switch on, it's a good idea to see how many mains sockets you have around, and where you are going to house everything. When you are in full control of your Spectrum you will need three mains sockets, one for the Spectrum, one for the cassette recorder, and one for the TV receiver. Most houses have desperately few sockets fitted, so you will find it worthwhile to buy or make up an extension lead that consists of a three or four-way socket strip with a cable and a plug (Figure 1.4). This avoids a lot of clutter – you don't want to bring your Spectrum crashing to the floor when you trip over a cable. Don't rely on the old fashioned type of three-way adaptor – they never produce really reliable contacts. The Spectrum has no on/off switch, and you should *always* take out the mains plug after you have finished a computing session. The noise from the power-pack should be a useful reminder!

When you have the essential equipment to start computing, consisting of the Spectrum keyboard, power supply, and TV (or monitor), quite a lot of flat surface is needed. Later on, you will probably want to add a printer, possibly disk drives and other extras which make the difference between having a computer *system* and just having a computer. All of this needs space, and the best way that I have found of organising this is one of the computer stands made by Selmor (Figure 1.5). If you aren't at that stage yet, then a good-sized desk or table will have to suffice for the time being. Computing is like Hi-Fi – there's always something else that you can buy, and there's never a time when you can truthfully say that you've done everything.

With everything housed and connected up, you now have to get used to some do's and don't's. The Spectrum uses an ordinary cassette recorder for recording and replaying its programs. This cassette recorder does not come with the computer, because most households have a cassette recorder – it doesn't have to be a special type. In addition, cassettes are cheap and easy to find, unlike the special discs you need on some other machines. You *do* need, however, to be able to adjust the cassette recorder to suit any tapes that you buy, or the tapes that come with the machine. Advice on that action follows later in this Chapter.

The next step, then, is to switch on the TV receiver and the Spectrum. Unless you are exceptionally lucky, or using a monitor, you will probably see nothing appear on

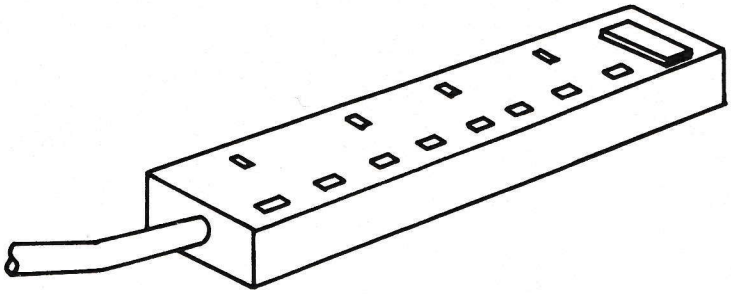


Figure 1.4 A four-way socket strip which avoids the use of the old-style adapters.

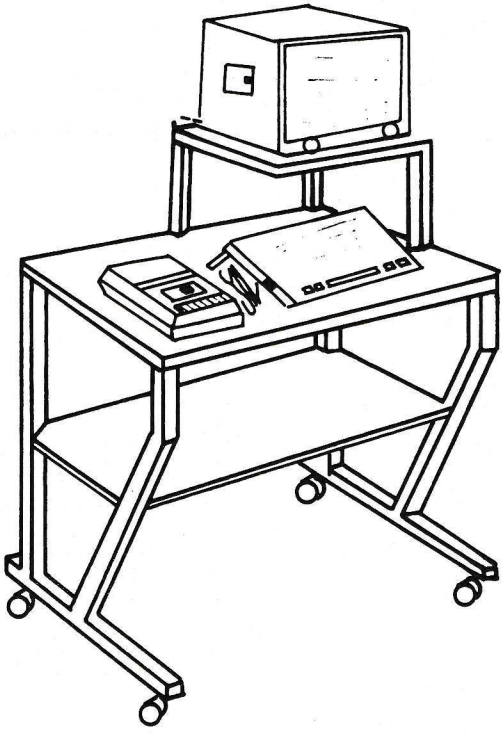


Figure 1.5 Using a *Selmor* stand to house all the bits and pieces of a Spectrum 128 computer system.

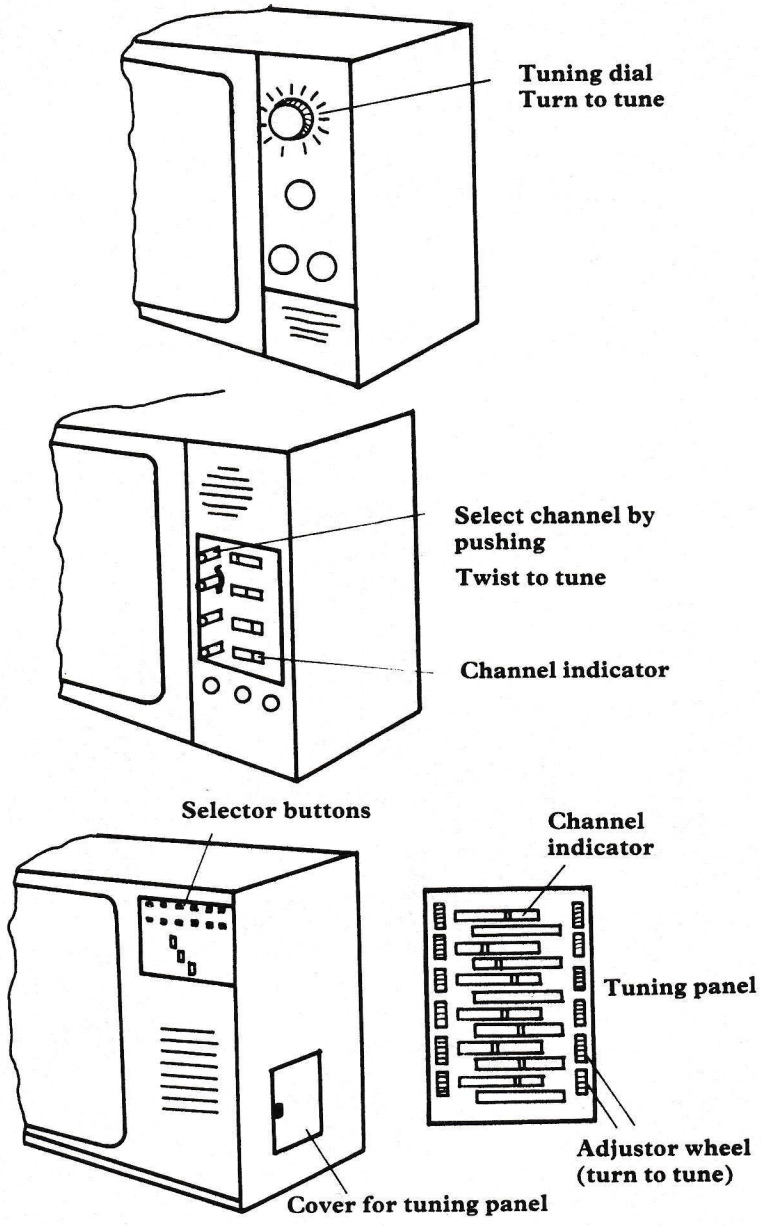


Figure 1.6 TV tuning controls. (a) Single dial, as used on black-and-white portables, (b) four-button type, (c) the more modern touch-pad or miniature switch type.

the TV screen. This is because a TV receiver has to be tuned to the signal from the Spectrum. Unless you have been using a video cassette recorder, and the TV has a tuning button that is marked 'VCR' it's unlikely that you will be able to get the Spectrum tuning signal to appear on the screen of the TV simply by pressing tuning buttons. The next step, then, is to tune the TV to the Spectrum's signals. The Spectrum can produce a test-signal for this purpose. You get this signal either by holding down the BREAK key as you switch the computer on, or if the machine is already switched on, by holding down the BREAK key and pressing the tiny RESET button at the left hand edge of the machine, just along from the input marked MIC. The test signals consist of a set of coloured bars, with the year 1986 printed in each bar, and a sound signal.

Figure 1.6 shows the three main methods that are used for tuning TV receivers in this country. The simplest type is the dial tuning system that is illustrated in Figure 1.6a. This is the type of tuning system that you find on black/white portables, and to get the Spectrum 128's test signal on the screen, you only have to turn the dial. If the dial is marked with numbers, then you should look for the signal somewhere between numbers 30 and 40. If the dial isn't marked, which is unusual, then start with the dial turned fully anti-clockwise as far as it will go, and slowly turn it clockwise until you see the Spectrum test signal appear.

When you can see the bars of the test signal appear, turn the dial carefully, turning slightly in each direction until you find a setting in which the words are really clear. Turn up the volume control of the receiver so that you can hear the sound signal, because this also should be clear. On a colour TV receiver the edges of the coloured bars may never be particularly clear, but get them steady at least and as clear as possible.

The older types of colour and B/W TV receivers used mechanical push-buttons (Figure 1.6b) which engage with a loud 'clonk' when you push them. There are usually four to six of these buttons, and you'll need to use a spare one which for most of us means the fourth or sixth one. Push this one fully in. Tuning is now carried out by rotating this button. Try rotating anti-clockwise first of all, and don't be surprised by how many times you can turn the button before it comes to a stop. If you tune to the Spectrum's signal during this time, you'll see the message on the screen. If you've turned the button all the way ant-clockwise and not seen the tuning signal, then you'll have to turn it in the opposite direction, clockwise, until you do. If you can't find the Spectrum signal at any setting, check that you have connected the cables to the TV aerial socket correctly. If you are using the PANDA adaptor, you can push one of the other tuning buttons to check that you can receive normal TV signals. If you can, there's nothing wrong with the TV, so switch back and try again to find the Spectrum signal.

Modern TV receivers are equipped with touch pads or very small push-buttons for selecting transmissions. These are used for selection only, not for tuning. The tuning is carried out by a set of miniature knobs or wheels that are located behind a panel which may be at the side or at the front of the receiver (Figure 1.6c). The buttons or touch pads are usually numbered, and corresponding numbers are marked on the tuning wheels or knobs. Use the highest number available (usually 6 or 12), press the pad or button for this number, and then find the knob or wheel which also carries this number. Tuning is carried out by turning this knob or wheel. Once again, you are looking for a clear picture on the screen and the sound from the loudspeaker. On this type of receiver, the picture is usually 'fine-tuned' automatically when you put the cover back on the tuning panel, so don't leave it off. If you do, the receiver's circuits that keep it in tune can't operate, and you will find that the tuning alters, so

that you have to keep re-tuning. The Spectrum should give a good picture on practically any TV receiver. I tried it with several, and even my Philips portable colour TV, which does not work well with computers, gave a reasonably good picture with the Spectrum. If your TV exhibits faults like a shaking picture, or very blurred colours, then check the tuning carefully. If the faults persist, and the TV is correctly tuned, you will have to contact the service agents for the TV – or use a different model!

When you have tuned the TV receiver to your satisfaction, press the little RESET button on the left-hand side of the keyboard, next to the MIC socket. This will return you to the main Spectrum menu that you normally see when you switch the machine on. The menu allows the choice of `Tape loader`, `128 BASIC`, `Calculator`, `48 BASIC`, or `Tape Tester`. For setting up the machine with its cassette recorder, you now need the Tape Tester, and you select this by pressing the key that is marked with a down-arrow (two keys to the right of the space bar). Each press of this key takes you down to the next item on the screen menu, and if you overshoot the display starts at the top again. When the shaded bar is over Tape Tester, then you press the L-shaped ENTER key on the right-hand side of the keyboard. This brings up a display which has a blue bar one third of the way down the screen. The bar is marked with squares that indicate tape signal strength, and the aim is to put in a tape and adjust the volume control of the cassette recorder until the signal strength is a maximum, but with the volume control as low as possible.

Cassette recorder adjustment

The next thing is to connect the cassette recorder – if you haven't already done this. The cable for the cassette recorder uses plugs called *jack-plugs*, and these are used on all the low-priced cassette recorders. You can't use the Hi-Fi type of recorder that has a different type of socket. The jack-plugs are colour-coded, one grey and the other black, and it doesn't matter how you use the colours as long as you match them up correctly. Suppose, for example, that you put the black plug at one end of the cable into the socket that is marked MIC on the Spectrum, at the left-hand side of the case. This means that the black plug at the other end of the cable has to be put into the MIC socket of the cassette recorder. Some cassette recorders use a symbol of a circle and a line for microphone. The other plug, the grey one, goes into the EAR socket of the Spectrum, and the EAR socket of the recorder. Once again, this may be marked with a symbol – a drawing of an ear – instead of the word.

Once the connections have been made, you can plug in the mains connector of the cassette recorder. If you normally use your cassette recorder with batteries, then it's better to get the mains cable out for this work, because batteries won't last long when you are loading programs, and the recorder works more steadily when the mains supply is used. With the Tape Tester selected on the computer, put in one of the tapes that comes with the computer, plug in the EAR plug, and press the PLAY key of the recorder. Nothing will happen for a few seconds, and then you will see a pale square appear in the bar that indicates signal strength. Adjust the volume control of your cassette recorder so that the square appears as far over to the right as possible. Don't worry if you can only get it half-way over, this is, in fact, quite good. You'll find that as you increase volume, the square moves over quite a long way, but after that, increasing the volume control level makes the square move no further. Leave the volume control setting at where the big change ends. If you use full volume, you

WARNING

When you record your own programs, you must not have the EAR plug connected. Get into the habit of leaving this plug out until you have to put it in – that is for loading in a program from tape. A recording made with the EAR plug in place will not load in correctly. This is a fault that has plagued all Spectrum models, and the new machine seems to be even more sensitive to it. If you follow this rule, however, you will experience no trouble.

will probably find that tapes don't load correctly, so reduce the volume to the point where the blue square is still well over on its scale, but not too far back from the position at full volume. This sets up the volume control correctly, but it doesn't guarantee that all tapes will load correctly. The reason is that tapes are made on copying machines, and these machines differ slightly, and are very different from your own cassette recorder. The important difference is the angle at which the magnetic head of the recorder touches the tape. If your cassette recorder head is at a different angle, it won't make the best job of loading a tape. Appendix A shows how to alter this if you find that after setting up and testing, you can't load the tapes that come with the Spectrum. To load one of these tapes, you need now to press BREAK twice, and you will see the menu again. This time, press ENTER when the shaded bar is on the first line, Tape Loader. You will get a message on the screen, and you can now press the PLAY key on the cassette recorder to load the tape. Don't forget to wind the tape back first if you have been using it for testing, and don't be surprised at how long it takes to load. You have a lot of memory to fill, and it all takes time. Don't stop the recorder until the tape is at its end. For a commercial tape like this, the program will start whenever the tape loads, and you will hear the sound of the program through the loudspeaker of your TV receiver. If you are using a monitor, you will need to use the type with a built-in loudspeaker, and the correct connections to the cable, if you are to get the sound delivered. You can also take the sound output from the Spectrum to a separate hi-Fi or Music Centre if you can get your local supplier to make up the correct cables for this purpose.

Now to work . . .

Once you have achieved a tuned signal, set up your cassette recorder, and loaded one of the test tapes into your Spectrum, the business of mastering the use of the computer begins. You aren't forced to program the Spectrum for yourself, of course. You can spend all of your time running programs that other people have written without ever programming the machine. If you do this, though, it's a bit of a waste, looking at all these keys that you hardly ever use. After all, buying a computer like this one and not programming it is rather like buying a Ferrari Testarossa and getting someone else to drive it for you, leaving you to buy the petrol and pump up the tyres. To get into programming, you have to make the menu selection of 128 BASIC. If you have been in the middle of a game, you will have to follow the instructions for leaving the game, like typing QUIT for leaving the Never-Ending Story. For some games, you might have to press the RESET button to escape.

When you select the 128 BASIC option of the menu, you will see the screen clear, with the 128 BASIC message at the bottom, and at the top left-hand corner a flashing square. This square is called the *cursor*. It is used as a marker, and when you press a

key, a letter (or number, or whatever is marked on the key) will appear at the position of the cursor. The cursor will then move across the screen to the next position. It's important to note at this point that nothing that you can do just by pressing keys on the keyboard can possibly damage the Spectrum machine – the worst you can do is to lose a program that was stored in the memory. You can, however, damage the Spectrum by spilling coffee all over it, dropping it, or connecting it up to other circuits while the power is switched on. You can also damage tapes if you attempt to take them out while they are still running. *Always* switch off the computer, and everything that is connected to it when you insert or remove any of the plugs at the back.

Key-tapping time

It's time now to look at the keyboard, because the keyboard is the way that you pass instructions to the Spectrum. Several of the keys are used mainly if you want to program it like the older Spectrum, and aren't used in 128 mode. Since this book is about 128-mode, we'll leave out the keys that don't apply. If we ignore the keys at the left-hand top side, then most of the Spectrum 128 keys look like typewriter keys. The arrangement of letters and numbers is much the same as that of a typewriter and if you've ever used a typewriter, particularly an electric typewriter, then you should be able to find your way round the keyboard of the Spectrum pretty quickly. When you press any of the letter keys, you will see the letter appear on the screen. What you see is a 'lower-case', small, letter, not an 'upper-case', capital, letter. Just like a typewriter, the keyboard of the Spectrum normally gives you lower-case letters. If you want a capital, you need to press one of the two CAPS SHIFT keys as well as a letter key. Commands that you give to the computer in typed form can be in either lower-case (small letters) or upper-case (capitals); it makes no difference to the way that the computer treats them *except for some sound commands*. If you want capitals all the time, press the key that is marked CAPS LOCK. To return to the original system, you only have to press the CAPS LOCK key again. Unfortunately, there is no indicator light to show you whether CAPS LOCK is on or off – you just have to try it and see! Whatever the setting of the CAPS LOCK key happens to be, the keys which show symbols on them, like most of the number keys, will need the use of SYMBOL SHIFT to get the symbol above the letter and to the right. The

exceptions are the Q, W, and E keys. The symbols on these keys refer to 48K BASIC use, and to get these symbols you need to press the keys that give the separate parts of the symbol. For example, the symbol on the Q key is <=, and you get this by using SYMBOL SHIFT R then SYMBOL SHIFT L. The EXTEND MODE key is used only for a very few symbols, the square and curly brackets for example, and the method of getting these is explained in the 128 booklet. If you have graduated to the Spectrum 128 from the older models, you will have to forget all about the way that you used to type words by pressing combinations of keys, because the 128 BASIC works by typing words in the normal way. If you learn BASIC on the Spectrum 128, then you will be able to program other machines, because you will have learned how to use the keyboard correctly. If at any time you need to use the machine to enter a program in 48 BASIC, you can switch to this (but not while you are running 128 BASIC) and use the machine just as you did with the old models. You can do this by typing SPECTRUM (press ENTER), or by using the RESET button to return to the main menu, or by using the EDIT key, and selecting the `Exit` choice.

As well as the ordinary typewriter keys, there are a number of special keys which are not found on any typewriter. At the left-hand side of the keyboard, for example, you will find keys labelled GRAPH and EDIT. The GRAPH key is used to get the *shapes* that appear on the number keys. You press the GRAPH key once to make these keys give the shapes, and press it again to stop the action. You don't have to press the GRAPH key each time together with the number key in the way that you did with the CAPS or SYMBOL keys. The EDIT key is used to bring up a new menu, and that's something that we'll look at later.

Of the other keys on the left-hand side, the TRUE VIDEO and INVERSE VIDEO keys have no action when you are using 128 BASIC; they are there simply to allow 48 BASIC to be used. The DELETE key means what it says – pressing this key will wipe out the letter or digit (character) that is immediately to the left of the cursor. You can use the arrowed keys on each side of the space bar to move the cursor around the screen, but only where there is something printed on the screen. If you try to move the cursor to where the screen is blank, you will get a beep from the loudspeaker of the TV (did you turn up the volume enough to hear it?) as a reminder. By using these cursor keys and the DELETE key, you can make changes to whatever you have typed, the action that is called *editing*. The EDIT key is for more specialised editing actions, and we'll come to these later.

Now we need to take a look at some of the specialised keys on the right-hand side of the keyboard. The BREAK key, pressed once, will stop a program in BASIC from running. When you stop a program in this way, you can start it again, because the program is not wiped from the memory. The program is started again by typing the instruction CONTINUE and pressing the ENTER key. The message that you get on the screen if the screen is full is to the effect that typing CONT will cause the action to restart, but on the Spectrum you must type the full word, CONTINUE. Another much more drastic way to stop a program, such as a game, from running, is to use the RESET button at the left-hand side, but this always has the effect of wiping out any program from the memory.

The set of four keys that are marked with arrows, on either side of the space bar, are *cursor* keys. The more useful two are at the left of the long *space bar*, and the other two are at the right. The two at the left are used to correct mistakes in your typing, so that you can rub out a letter, replace one letter with another, or insert letters where you want. That's something else that we'll come back to later, in Appendix B.

The most important of all of the special keys, however, as far as we are concerned at the moment, is the key that is marked ENTER. This is in the position of the 'carriage return' key of an electric typewriter, but its action is not the same in all respects. Pressing the ENTER key is a signal to the computer that you have completed typing an instruction and that you now want the computer to obey it.If you are accustomed to using an electric typewriter, you will have to change some of your habits as far as this key is concerned. During the use of a typewriter, you would press the carriage return key each time you wanted to select a new line, with typing starting at the left-hand side of the new line. The ENTER key of the computer does rather more than this. If the material that you are typing into the Spectrum 128 takes more than one line on the screen, the machine will *automatically* select the next screen line for you. The ENTER key must *not* be used for this purpose. The ENTER key is used only when you want the machine to carry out a command or store an instruction, not simply when you want to use a new line. It will always provide a new line for you, however, and select a position at the left-hand side. The position where a letter or other character will appear when you press a key is indicated by the cursor.

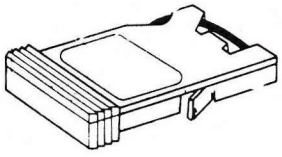
You will find that the action of each key repeats if you hold your finger on it, and this repeat action is quite fast. If you type a set of meaningless letters and then press ENTER, the computer usually responds with some message at the bottom of the screen such as:

```
2 Variable not found, Ø:1
```

This is because the computer is a simple machine. It can understand only a few words, the words that we call its *reserved words* or *instruction words*. If what you type does not include these words, or uses these words incorrectly, then this is an error of some kind as far as the computer is concerned. The words may make sense to you, but not to the computer. By way of revenge, you get an error message that you may not be able to understand at first!

Cassette tryout

The Spectrum 128 computer, like all others, stores instructions and other information in its memory – but only while the machine is switched on. Whenever you switch off your Spectrum, anything that was stored in its memory is instantly lost, and you can't get it back even if you switch on again very quickly. For that reason, we need to store the programs and the information that we need to use in programs in another form. The most useful forms are magnetic disks and magnetic tape. The Spectrum has at the time of writing no 'official' way of using magnetic disks, such as are used for business applications, but uses a cassette recorder instead. Several unofficial ways of connecting the Spectrum to disk systems have, however been advertised – the Opus one is well-known and widely used. At the time of writing, you can also still buy the Spectrum Microdrive system that uses special tape cartridges.



A Microdrive cassette

The computer has circuits that will convert the instructions of a program into signals, which can then be recorded on the tape. When these signals are replayed, another set of circuits will convert the signals back into the form of a program. In this way, the use of the cassette recorder allows you to record your programs on tape and to replay them again. Before you tackle the rest of this book, then, it's important to check now that you can use the recorder correctly to record and replay programs.

By this time, you will have used the recorder to put a program into the memory. This is called *loading*, and since your Spectrum comes with a set of two programs on cassettes, these are a convenient way of getting experience. Once you have gained some experience with loading the free tapes, you need to try making a recording to check the action. This also acts as another check on the settings of the cassette recorder, because the Spectrum is, unusually, more sensitive about its own recordings than about bought ones! Before you can make a recording to test the system, you need a program to record, and this involves some typing. This is easy if you have just switched the Spectrum on, but if you have been pressing keys at random, then it's a good idea to press the RESET button, or to switch off again, then on. The next step *always* is to remove the EAR plug at either end.

Type the number 10 (1 and then 0), and then the word `rem`. It doesn't matter whether you type `rem` or `REM`. Check that this looks correct, and then press the ENTER key. The effect of this is to place the instruction line 10 REM into the memory of the Spectrum. The beep that you hear as you press each key turns to a lower note after you press ENTER, showing that the machine is ready for another command. As you type the first digit, the character will be seen on the screen at the cursor position. When you press the ENTER key, the cursor moves to the next line down. At the same time, your command appears at the top of the screen, in the form:

```
10 REM
```

If you see a mistake as you type the line, just use the back space action, by pressing the DELETE key. This shifts the cursor backwards, and deletes the letter that the cursor is now over. You can then type the correct letter, which will replace the incorrect one. If this makes the line correct, pressing ENTER will enter it into the computer. If you have typed something that is incorrect, like `REN` or `RWM` then you will be warned by a beep, and the cursor will be placed to the left of the first incorrect letter – the computer will not accept anything incorrect that you type following a number. Now type the rest of the lines, as illustrated in Figure 1.7, remembering to press the ENTER key after you have completed typing each line. The numbers are called *line numbers*, and they are present for two reasons. One is to remind the computer that this is a program, the other is to guide it, because the computer will normally carry out instructions in the same order as the line numbers. You can check that your program looks correct by asking the computer to 'list' it. Listing means that the computer prints on the screen whatever you have stored in its memory. Using the line numbers ensures that the instructions are stored, and if you type `list`, and then press ENTER, you will see your program again. Don't be surprised to find that all the lower-case letters (like `rem`) have been converted to upper-case (like `REM`), because this is part of the action of the computer. Check from this *listing* that the program is like the printed version in Figure 1.7. The action of the Spectrum is such that you will see your program listing at all times unless the program is actually running, so that you don't usually need the command `LIST`.

Now make sure that the cassette recorder is ready, with a cassette in place and the EAR plug out. If you are using a brand new cassette make sure that it has been

```
10 REM
20 REM
30 REM
40 REM
```

Figure 1.7 A simple program for testing the cassette recording and replaying actions. Don't forget that EAR plug!

wound on far enough to ensure that the magnetic tape is showing. A completely rewound cassette has clear (or coloured) plastic tape, called a *leader* at the end, and you can't record on this. You need to wind this on a short distance until you can see the brown recording tape in place. If your recorder has a tape counter, you can wind on for a count of 5; the alternative is to push the body of a ball-pen into the centre of the empty tape-reel and turn the pen so that the tape is wound by hand to the correct position. When you are certain that the recording tape is wound on, replace the cassette in the recorder.

Now type:

```
SAVE "test"
```

– and press the ENTER key. The word SAVE is the instruction to the computer meaning that you want to save (record) a program onto a cassette. The 'test' part is a *filename* which the computer will use to recognise the program if it is asked to. You must place this name between quote marks (") – the quotes key is on the left of the space bar. Watch that you don't use the quotes key while you are pressing the CAPS SHIFT – this can produce odd effects, like displaying the copyright sign when you press the quotes key again! Normally, you will keep a number of different programs on each cassette, and you need the filenames to instruct the computer to load back the correct one. When you press the ENTER key, a screen message will ask you to start the recorder and then press any key. You should press the RECORD and PLAY keys of the recorder down firmly, and then press the space bar or the ENTER key of the computer. A few recorders need only the RECORD key pressed, but you will know what is needed for your own recorder. You will see patterns appear on the border of the screen as the program records, and you should not stop the recorder until you see the message `Ø OK, Ø:1` appearing at the bottom of the screen. Remember what the screen border looked like when this was happening, because a similar pattern of lines shows when the program is loading back in again.

Now comes the crunch. You have to be sure that the recording was O.K. Type `NEW` and press ENTER. This should have wiped your program from the memory. Just to make it look better, type `CLS` and press ENTER. This clears the screen, so that there will be no confusion. Now type `LIST` and press ENTER. Nothing should appear – LIST means put a list of the program instructions on the screen, and there shouldn't be any! You can now load the instructions in from the tape. First, put the EAR plug in place. There are two ways of loading the recording that you have made. One is to use the Tape-Loader item of the menu. When this has completed loading the program, the message `Ø OK Ø:1` will appear to show that loading is complete, and you then have to get back to the menu to select 128 BASIC to list the program. This is done by pressing BREAK to get the menu, and then selecting 128 BASIC. The alternative method is to keep with 128 BASIC, and type `LOAD""`, then press ENTER. You can then start the tape playing, and you will see the same screen display as you do when the tape-loader is used. When the final message appears, you can press ENTER, and this will list the program for you. This is the easier method if you want to stay with BASIC. Once you can reliably save programs on tape, and re-load them,

you can confidently start computing. When you have spent an hour or more typing a program on to the keyboard, it's good to know that a few minutes more work will save your effort on tape so that you won't have to type it again. You must, however, treat your cassettes with some care, avoiding magnets in particular. The table in Figure 1.8 should act as a useful reminder to you in this respect.

1. **Keep cassettes dry and in a cool place that is free from condensation.**
2. **Store and use cassettes well away from magnets. This includes loudspeakers, TV receivers and electric motors. Don't leave a cassette lying on top of the loudspeaker of the cassette recorder!**
3. **Never touch the tape with your fingers.**
4. **If the tape jams, take the cassette out, hit it against your hand, and then try to fast-wind in each direction.**
5. **Never use the first two feet of tape at either end of a cassette.**
6. **Never keep a tape stored for too long without playing it or at least fast-winding it.**

Figure 1.8 A checklist to ensure long life from your cassettes.

Chapter 2

Putting it all on the screen

Chapter 1 will have broken you in to the idea that the Spectrum 128, like practically all computers, takes its orders from you when you type them on the keyboard. You will also have found that an order is obeyed when the ENTER key is pressed. You will by now have used the command LIST which prints your program instructions on to the screen, and noticed that you get a listing automatically when a program is not running. There are two other useful points that you need to know before we go much further. One is that you can clear the screen by putting `CLS` (or `cls`) into a program. This also works as a direct command, by typing `CLS` or `cls` and then pressing the ENTER key, but your listing will appear again when you press another key afterwards. The instructions or messages at the bottom of the screen are *not* erased in this way, however. As your familiarity with the computer keyboard increases, you will want to make use of the editing commands, and these are explained in Appendix B.

Now there are two main ways in which you can use a computer. One way is called *direct* mode. Direct mode means that you type a command, press ENTER, and the command is carried out at once. This can be useful, but the more important way of using a computer is in what is called *program mode*. In program mode the computer is issued with a set of instructions, with a guide to the order in which they are to be carried out. A set of instructions like this is called a program. The difference is important, because the instructions of a program can be repeated as many times as you like with very little effort on your part. A direct command, by contrast, will be repeated only if you type the whole command again, and then press ENTER. The set of command words that can be used, along with the rules for using them, make up what is called a *programming language*. The Spectrum 128 provides you with a well-tried and very practical version of the most commonly-used of all programming languages for small computers, BASIC. BASIC is short for 'Beginners All-purpose Symbolic Instruction Code', and it was originally devised for teaching purposes. Since then, it has developed into a useful language in its own right. The version of BASIC that your Spectrum 128 uses is, however, enriched by a lot of extra commands.

An important point about all computer commands, whether they are direct commands or program instructions, is that they have to be in a precise form. This is

particularly true of the Spectrum 128, and if you have programmed a ZX-81 or Spectrum previously, you will have to get rid of some old habits. To start with, when you use 128 BASIC, you need to type each command word in full. This might sound as if it would take a lot longer, but it's no more difficult than trying to work out what combination of keys to use with the older Spectrum, and it's a better preparation for using other machines. Another surprise with the Spectrum 128 is the Calculator Mode. This allows you a special form of direct commands that use the machine as a calculator. There is an important difference between using the Spectrum 128 as a calculator and using an actual calculator – the Spectrum 128 does not accept the = sign in a line. You can, for example, type `3+2` and press ENTER to get `5`, but you can't use `3+2=` and then press ENTER. With that reservation, the Calculator mode is very useful, and you can jump between this mode and 128 BASIC *without* losing any program that you have in the memory. To jump from one to the other, press the EDIT key, and select the `EXIT` option. This gets you to the main menu and you can then pick 128 BASIC or Calculator as you please, depending on which way you want to move. For the rest of this book, however, we'll be concerned only with the machine in its 128 BASIC mode.

Let's now take a look at the difference in 128 BASIC between a direct command and a program instruction. If you want the computer to carry out the direct command to add two numbers, 1.6 and 3.2, then you have to type:

`PRINT 1.6 + 3.2` (and then press ENTER)

You have to start with PRINT (or print), because a computer is a dumb machine, and it obeys only a few set instructions. Unless you use the word PRINT, the computer has no way of telling that what you want is to see the answer on the screen. It doesn't recognise instructions like `GIVE ME` or `WHAT IS`, only a few words that we call its *reserved words* or *instruction words*. PRINT is one of these words. So that you can recognise these reserved words more easily in this book, they are printed in upper-case (capital) letters from here on, words in `computer type` represent things that are typed into, or displayed on, the computer. You know by now, however, that you can type them in either upper-case or lower-case. You will *always* see them in upper-case when you LIST your program, whether on the screen or on a printer. There does not have to be a space following the 'T' of PRINT, but it looks better and makes the line easier to read if there is one. You do not need to have spaces between the '6' and the '+' or the '+' and the '3'.

When you press the ENTER key after typing `PRINT 1.6 + 3.2`, the screen shows the answer, `4.8`. This answer is not shown in the same place as your typed command, however. If you have just cleared the screen before typing, the answer will appear at the top left-hand corner. If there is anything else printed on this main part of the screen, the answer of `4.8` will appear on the next line. Once a direct command has been carried out, however, it's finished. A program does not work in the same way. A program is typed in, but the instructions of the program are not carried out when you press the ENTER key. Instead, the instructions are stored in the memory, ready to be carried out as and when you want, or recorded for use at some other time. The computer needs some way of recognising the difference between your direct commands and your program instructions. On computers that use the BASIC language, this is done by starting each program instruction with a line number. This must be a positive whole number – a positive integer. This is why you can't expect the computer to understand, in 128 BASIC mode, an instruction like `5.6 + 3=`, it takes the 5 as being a line number, and the rest doesn't make sense.

```
10 PRINT 5.6+6.8
20 PRINT 9.2-4.7
30 PRINT 3.3*3.9
40 PRINT 7.6/1.4
```

Figure 2.1 A four-line arithmetic program.

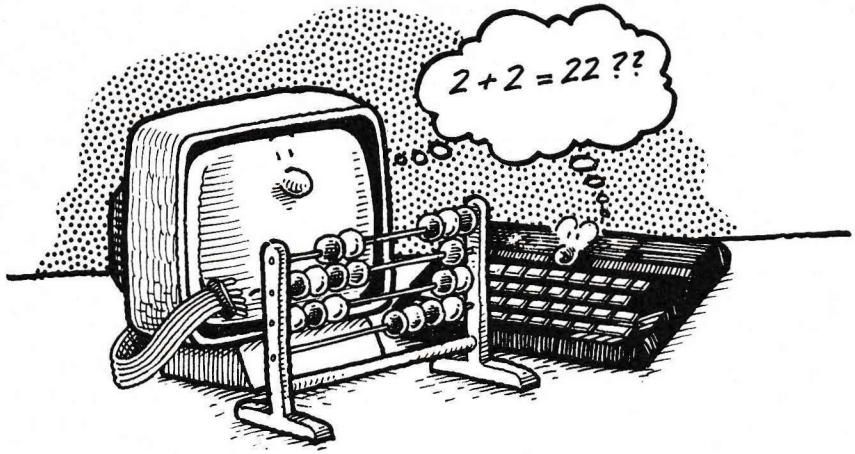
Let's start programming, then, with the arithmetic actions of *add*, *subtract*, *multiply* and *divide*. Computers aren't used all that much for calculation, but it's useful to be able to carry out calculations now and again. Figure 2.1 shows a four-line program which will print some arithmetic results. Take a close look at this, because there's a lot to get used to in these four lines. To start with, the line numbers are 10, 20, 30, 40 rather than 1, 2, 3, 4. This is to allow space for second thoughts. If you decide that you want to have another instruction between line 10 and line 20, then you can type the line number 15 , or 11 or 12 or any other whole number between 10 and 20, and follow it with your new instruction. Even though you have entered this line out of order, the computer will automatically place it in order between lines 10 and 20. If you number your lines 1, 2, 3 then there's no room for these second thoughts though you can change line numbers if you have to by using the editing command RENUMBER.

The next thing to notice is how the number Ø on the screen is slashed across. This is to distinguish it from the letter O. The computer simply won't accept the 0 in place of O, nor the O in place of 0, and the slashing makes this difference more obvious to you, so that you are less likely to make mistakes. If you have worked with an old typewriter with no '1' (one) key, you may have become used to typing a letter 'l' instead, and this is another habit that you *must* lose! The zero that you see on the keyboard is also slashed and is on a different key, and is differently shaped. Type some zero's and O's on the screen so that you can see the difference.

Now to more important points. The minus sign is obtained by pressing SYMBOL SHIFT and J, not SYMBOL SHIFT and 0, which is an underline dash. The star or asterisk symbol in line 30 is the symbol that the Spectrum 128 (and all other computers) uses as a multiply sign. Once again, we can't use the 'x' that you might normally use for writing multiplication because this is a letter. There's no divide sign on the keyboard either, so the Spectrum 128, like all other small computers, uses the forward slash (/) sign in its place. This is the diagonal line on the same key as the V, obtained by using SYMBOL SHIFT and V, *not* the forward slash obtained by using EXTEND MODE, SYMBOL SHIFT and then CAP SHIFT and D.

So far, so good. The program is entered by typing it, just as you see it. You don't need to leave any space between the line number and the P of PRINT, because the Spectrum 128 will put one in for you when it displays the program on the screen. You will have to press the ENTER key when you have completed each instruction line, before you type the next line number. You should end up with the program looking as it does in the illustration. When you have entered the program by typing it, it's stored in the memory of the computer in the form of a set of code numbers. There are two things that you need to know now. One is how to check that the program is actually in the memory, the other is how to make the machine carry out the instructions of the program.

The first part is dealt with using the command LIST that you know already. You will then see how the computer has printed the items of the program on the screen, with spaces between the line numbers and the instructions. To make the program



operate, you need another command, RUN. Press ENTER to get the machine ready for a command, then type RUN, then press the ENTER key , and you will see the instructions carried out. To be more precise, you will see:

```
12.4
4.5
12.87
5.4285714
```

That last line should give you some idea of how precisely the Spectrum 128 can carry out this type of arithmetic. A calculator will give several more places of decimals, but seven places is usually enough for any practical purposes. You'll notice, by the way, that the results are not printed over the program listing, because the screen is automatically cleared when the program runs. You can get back to the listing by pressing the ENTER key again. you may find that the cursor appear between lines of the listing, but that's no problem. The only thing to watch for is that you don't type a command when the cursor is actually over on of the lines of the listing.

When you follow the instruction word PRINT with a piece of arithmetic like 2.8 * 4.4, then what is printed when the program runs is the *result* of working out that piece of arithmetic. The program *doesn't*, for example, print `2.8 * 4.4`, just the result of the action 2.8 × 4.4.

Now this is useful, but it's not always handy to get a set of answers on the screen, especially if you have forgotten what the questions were. The Spectrum 128 allows you a way of printing anything that you like on the screen, exactly as you type it, by the use of what is called a *string*.

Figure 2.2 illustrates this principle. In each line, some of the typing is enclosed between quotes (") and some is not. The semicolons are *very* important, and you must not omit them in the way that you can on a lot of other computers. Enter this short program and run it. Can you see how very differently the computer has treated the instructions? Whatever was enclosed between quotes has been printed exactly as you typed it. Whatever was not between quotes is worked out, so that the first line, for example, gives the unsurprising result:

```
2+2=4
```

```
10 PRINT "2+2=";2+2
20 PRINT "2.5*3.5=";2.5*3.5
30 PRINT "9.4-2.2=";9.4-2.2
40 PRINT "27.6/2.2=";27.6/2.2
```

Figure 2.2 Using quote marks to show that characters have to be printed on the screen exactly as typed.

Now there's nothing automatic about this. If you type a new line:

```
15 PRINT "2+2=";5*1.5
```

then you'll get the daft reply, when you RUN this, of:

```
2+2= 7.5
```

The computer does as it's told and the result is what you told it to do. Only a lunatic would believe that computers could take over the world! This is a good point also to take notice of something else. The line 15 that you added in this example has been fitted into place between lines 10 and 20 – LIST if you don't believe it. No matter in what order you type the lines of your program, the computer will sort them into order of ascending line number for you.

With all of this accumulated wisdom behind us, we can now start to look at some other printing actions. PRINT, used alone in this way always means print on to the TV screen. For activating a paper printer (hard copy, it's called), there's a separate variety of PRINT instruction which is obtained from the menu that you get by using the EDIT key. This variety of PRINT is not useful to you unless you have a printer connected, and Chapter 11 illustrates how to use a serial printer with the Spectrum 128. Similarly, the command LLIST is for making a listing appear on the printer, and you can't use it correctly unless you have a printer connected.

Now try the program in Figure 2.3. You can try typing the lines in any order that you like, to establish the point that they will be in line-number order when you press ENTER after each line. When you RUN the program, the words appear on three separate lines. This is because the instruction PRINT doesn't just mean print-on-the-screen. It also means 'take a new line', and start at the left-hand side! You will also find, for a longer piece of writing on the screen, that if the words on the screen reach the bottom line of the display section, then all the lines appear to move up, and the top line disappears. This action is called *scrolling*, and it's the way that the machine deals with displaying lots of lines on a screen which holds only 22 lines, not counting the bottom lines which are used for looking at commands that you are typing. The machine will hold up scrolling for you so that you can look at a set of lines at a time, then move on by pressing the Y key in response to the question scroll? at the bottom of the screen.

```
10 PRINT "This is"
20 PRINT "the remarkable"
30 PRINT "Spectrum 128 compute
r."
```

Figure 2.3 Using the PRINT instruction to place words on the screen.

```
10 PRINT "This is ";
20 PRINT "the remarkable ";
30 PRINT "Spectrum 128 compute
r."
```

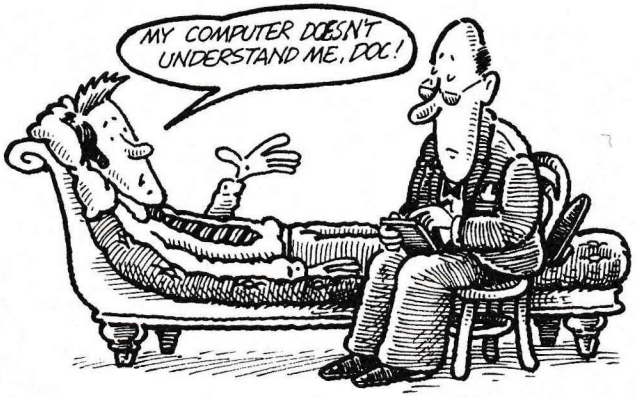
Figure 2.4 The effect of semicolons to prevent a new line from being taken.

Now the action of selecting a new line for each PRINT isn't always convenient, and we can change the action by using punctuation marks that we call print modifiers. Start this time by acquiring a new habit. Type NEW and then press the ENTER key. This clears the old program out, and you are returned to the main menu, so that you can select 128 BASIC again for the next sample. If you don't use the NEW action, there's a chance that you will find lines of old programs getting in the way of new ones. Each time you type a line, you delete any line that had the same line number in an older program, but if there is a line number that you don't use in the new program it will remain stored. In Figure 2.2, for example, the line 15 that you added later would be left in store even when you typed a new line 10 and a new line 20. If you are using a printer with your Spectrum 128, you will have to remember that the EDIT key can be used to print the lines of a program, but that a different command, *LPRINT* has to be used in the lines of the program if you want the program to produce printed output. In addition, each time you use NEW with a printer attached, you will have to set up your printer link again – see Chapter 11 for details.

Now try the program in Figure 2.4. There's a very important difference between Figure 2.4 and Figure 2.3, as you'll see when you RUN it. The effect of a semicolon following the last quote in a line is to prevent the next piece of printing starting on a new line at the left-hand side. When you RUN this program, most of the words appear in one line. It would have been a lot easier just to have one line of program that read:

```
10 PRINT "This is the remarkable Spectrum 128 Computer"
```

to do this, but there are times when you have to use the semicolon to force two different print items on to the same line. We'll look at that sort of thing later in program examples.



Rows and columns

Neat printing is a matter of arranging your words and numbers into rows and columns, so we'll take a closer look at this particular art now. To start with, we know already that the instruction PRINT will cause a new line to be selected, so the action of Figure 2.5 should not come as too much of a surprise. Lines 10 and 20 contain a novelty, though, in the form of two instructions in one line. The instructions are separated by a colon (:), and you can, if you like, have several instructions following one line number in this way, taking several screen lines. The only practical limit to this is that it makes your instructions too hard to read if you put too many instructions together in this way. In a 'multi-statement' line of this type, the Spectrum 128 will deal with the different instructions in a left-to-right order.

```
  10 CLS : PRINT "This is the Sp
ectrum 128"
  20 PRINT : PRINT
  30 PRINT "Ready to work for yo
u"
```

Figure 2.5 Clearing the screen with the CLS instruction, and using multi-statement lines.

The instruction CLS should not surprise you either – this clears the screen, and makes the printing start at the top left-hand corner. It's the same action as the CLS direct command, but done automatically within the program. As it happens, it's not needed as the first command in a Spectrum 128 program because the screen is cleared automatically anyhow, but it's useful to bear in mind for any time that you might need to clear the screen in the middle of a program.

Another point about Figure 2.5 is that line 20 causes the lines to be spaced apart. The two PRINT instructions, with nothing to be printed, each cause a blank line to be taken. There are other ways of doing this, as we'll see, but as a simple way of creating a space, it's very handy.

```
  10 PRINT 1,2
  20 PRINT 1,2,3,4
  30 PRINT "one","two"
  40 PRINT "one","two","three","
four"
  50 PRINT "This item is longer"
,"two"
  60 PRINT 1,2,3,4,5
```

Figure 2.6 How the comma causes words or numbers to be placed into two columns.

Figure 2.6 deals with columns. Line 10 is a PRINT instruction that acts on the numbers 1 and 2. When these appear on the screen, though, they appear spaced out just as if the screen had been divided into two columns. The mark which causes this effect is the comma, and the action is completely automatic. The comma is on the key next to the right-hand SYMBOL SHIFT and if you use the apostrophe on key 7, you will not get the same effect! The two look rather alike on the keyboard, but completely different on the screen. The use of the apostrophe causes a new line to be taken, which is not the same as the effect of the comma. As line 20 shows, you can get only two columns, each one of which allows room for sixteen characters. Anything that you try to get into a third column will actually appear on the first column of the next line down. The action works for words as well as for numbers, as lines 30 and 40 illustrate. When words are being printed in this way, though, you have to remember that the commas must be placed *outside* the quotes. Any commas that are placed inside the quotes will be printed just as they are and won't cause any spacing effect. You will also find that if you attempt to put into a column something that is too large to fit, the long phrase will spill over to the next column, and the next item to be printed will be at the start of the next line. Line 50 illustrates this – the first phrase spills over from column 1 all the way into column 2, and the word 'two' is printed starting at column 1 on the next line. Line 60 shows what happens if you keep using commas – the columns just take up the same positions on the next line in turn.

Commas are useful when we want a simple way of creating two columns. A much more flexible method of placing words on the screen exists, however.

This is programmed by using the command word AT, and it allows you to place words on the screen at any position, and also in any order. Normally, when you PRINT on to the screen, a PRINT instruction causes the computer to take a new line and you cannot go back to the previous one. By using AT, you can place words and numbers where you like, and even have one line replace another if you want to.

AT must follow PRINT, and in turn must be followed by two numbers. Of these, the first number is a line number. You can print up to 22 lines on a screen, and so these line numbers range from 0 (top of screen) to 21 (bottom of screen). The second number is a column number. You can print 32 characters (letters, numbers, punctuation marks) in a line across the screen of the Spectrum 128. Each of these characters is in one column, because characters in all of the other lines are also spaced out in the same way. We can use the column number, then, to represent the position of a character along a line. The number can range from 0 (left-hand side) to 31 (right-hand side) – a total of 32 columns. You *must* type a semicolon following the second of the numbers, before you put in whatever is to be printed.

An example would certainly help here. Take a look at Figure 2.7. Line 10 clears the screen, and line 20 introduces the first PRINT AT. This is 0,0, meaning that printing will start on the top left-hand corner of the screen. The `T` of `TOP LEFT` will therefore be placed at the top left corner of the screen, as it would anyhow just following a CLS. The next line then causes a pause. The word PAUSE, followed by a number, causes a time delay . The number is the delay time in units of 1/50 seconds (1/60 in the U.S. version), so that the number 50 would give a delay of one second, 100 gives two seconds and so on.

The next AT is followed by 21,20; so that the phrase `BOTTOM RIGHT` is placed with the `T` of `RIGHT` in the bottom right-hand corner of the screen. How was this calculated? The answer is by counting the `T` of `RIGHT` as column 31, then counting back 30, 29, 28 and so on until I reached the `B`, which was at 20. This is to be on the bottom line, number 21. With this much information now, you can see how the rest of the words

```
 10 CLS
 20 PRINT AT 0,0;"Top left"
 30 PAUSE 100
 40 PRINT AT 21,20;"Bottom righ
t"
 50 PAUSE 100
 60 PRINT AT 21,0;"Bottom left"
 70 PAUSE 100
 80 PRINT AT 0,22;"Top right"
 90 PAUSE 100
100 PRINT AT 11,13;"Middle"
```

Figure 2.7 Using the 'print modifier' word AT, which allows printing to be placed anywhere on the screen.

have been put into position. Because of the pauses, you can see that we are not following the normal order of left-to-right, top-to-bottom for printing.

How did I position the word MIDDLE at the centre of a line in Figure 2.7? This is done by calculating the correct numbers for the AT instruction. The method is shown in Figure 2.8. You have to count up the number of characters that you want to print centered. By characters, I mean letters, digits, spaces and punctuation marks. You then subtract this from 32, and divide the result by 2. Take the whole number part of the answer – forget about any half left over – and this is then the correct column number to use with AT. In this example, we also wanted to place the word in the central line. With an even number of lines, there is no central line, so we can use either line 11 (too low) or line 10 (too high). In a later Chapter, you will see that we can use letters in place of numbers in the AT (and other) instructions. This allows us to centre words without all the fuss of counting letters – but that's more advanced programming than we should be thinking about at this point!

Figure 2.9 is a map that you will find useful for placing words where you want them on the screen. Meanwhile, there's more to learn about printing characters. You will find that when you first start programming for yourself, the appearance of words on the screen leaves a lot to be desired. You will, in particular, find that words are split from one line to the next in a very untidy way. The Spectrum 128 offers you help with this in the shape of the apostrophe character, which is on the 7 key, used with SYMBOL SHIFT. If you place this character following a PRINT, but not between quotes, it will cause a new line to be taken. This allows you to organise all of your printing in one command. If you have a line that starts normally with PRINT", then you type words until you can see that the next word that you type will go to a column beyond the one which contains the quote marks. At this point, you can put in quote marks, then the apostrophe sign, then more quotes – and continue! Provided you never type a word across the column which contains the quote mark for the previous line, your printing will look neat. The trouble with this method is that you don't see the lines correctly set out on screen until after you press ENTER. If you count out the characters on the first line, however, its easier to do. Take a look, for example, at the listing of Figure 2.10.

There's one more important way of changing how items are printed on to the screen. This uses the command word TAB, following PRINT. TAB is a simpler version of AT, using a column number only. By using TAB with a column number of 0 to 31, you can make an item print anywhere in a line. The TAB control is not quite so good as that of AT, however, because you can TAB only left to right. Once you have made

1. **Count number of characters in the title, including spaces.**
2. **Subtract this number from 32 if it is even, from 31 if it is odd.**
3. **Divide the result by 2.**
4. **Use the result as the TAB number, or as the column number in AT. *Alternatively*, divide the number of characters by 2, and then subtract the result from 16.**

Figure 2.8 How a phrase can be centered on a line. This can be used with either TAB or AT, but when you use AT, the row number will be needed as well.

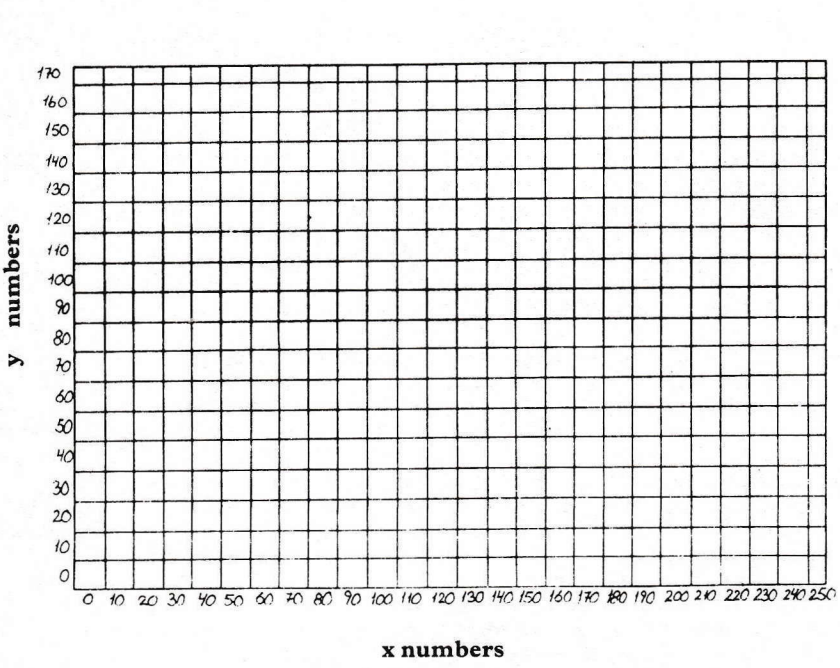


Figure 2.9 The AT map, which shows how the AT numbers correspond to positions on the screen.

```
  10 PRINT ''"Example"''
  20 PRINT "This uses the apostr
ophe in"'"order to arrange the l
ines"'"of print
better"
```

Figure 2.10 Using the apostrophe to arrange print into new lines without using an extra PRINT instruction. Note that the word better should follow immediately after print. The new line here has been caused by the action of my printer, which takes a new line after 80 characters.

```
  10 CLS
  20 PRINT TAB 5;"tab 5"
  30 PRINT TAB 5;"tab 5";TAB 15;
"tab 15";TAB 25;"tab 25"
```

Figure 2.11 How TAB is used to position the cursor in a program. Note that you can have more than one TAB in a line.

an item print at the right-hand side of a column, you can't make the next one print at the left. Figure 2.11 illustrates TAB in use so that you can see how it fits into the PRINT action. Notice that you can have more than one TAB in a PRINT line, but you have to make use of semicolons to separate the parts of the line. Notice that you must type `PRINT TAB` – you will get an error message if you try to use `PRINTTAB`. Now you have complete print control!

Chapter 3

A variable feast

So far, our computing has been confined to printing numbers and words on the screen. That's one of the main aims of computing, but we have to look now at some of the actions that are usually needed before anything is printed. One of these is called *assignment*. Take a look at the program in Figure 3.1. Type it in, run it, and contrast what you see on the screen with what appears in the program. The first line that is printed is line 20. What appears on the screen is:

```
2 times 23 is 46
```

but the numbers 23 and 46 don't appear in line 30! This is because of the way we have used the letter x as a kind of code for the number 23. The official name for this type of code is a 'variable name'.

Line 20 assigns the variable name x, giving it the value of 23. Assigns means that wherever we use x, *not* enclosed by quotes, the computer will operate with the number 23 instead. Since x is a single character and 23 has two digits, that's a saving of space. It would have been an even greater saving if we had assigned x differently, perhaps as `LET x=2174.3256`, for example. Line 30 then proves that x is taken to be 23, because wherever x appears, not between quotes, 23 is printed, and the *expression* `2*x` is printed as 46. We're not stuck with x as representing 26 for ever, though. Line 40 assigns x as being 5, and lines 50 and 60 prove that this change has been made.

```
  10 CLS
  20 LET x=23
  30 PRINT "2 times ";x;" is ";2
*x
  40 LET x=5
  50 PRINT "x is now ";x
  60 PRINT "and twice ";x;" is "
;2*x
```

Figure 3.1 Assignment in action. The letter x has been used in place of a number.

```
  10 CLS : LET n$="Spectrum 128"
  20 LET f$="The remarkable": LE
T L$="computer system"
  30 PRINT f$;" ";n$;" ";'L$
  40 PRINT "This uses the ";n$
  50 PRINT f$;" ";n$;'"in action
"
```

Figure 3.2 Using string variables. These are distinguished by the use of the dollar sign. The string variable name can consist of only one letter.

```
  10 CLS : LET B$="The new compu
ter"
  20 LET P$="that brings more po
wer."
  30 PRINT "Spectrum 128 _";
  40 PRINT B$'P$
  50 PRINT B$;" ";P$
```

Figure 3.3 Illustrating the use of string variable names for phrases of several words.

```
  10 LET a=2: LET b=3
  20 LET a$="2": LET b$="3"
  30 CLS
  40 PRINT a;" times ";b;" is ";
a*b
  50 PRINT "numbers are ";a$;" a
nd ";b$
  60 PRINT a$;" times ";b$;" is
impossible!"
```

Figure 3.4 String and number variables might look alike when they are printed, but they are different!

That's why we call x a *variable* – we can vary whatever it is we want it to represent. Until we do change it, though, x stays assigned as it was. Even after you have run the program of Figure 3.1, then providing you haven't added new lines or deleted any part of it, you can type PRINT x , and pressing ENTER will show the value of x on the screen. The listing also shows that this action of assignment must be carried out with the word LET coming just before the variable.

This very useful way to handle numbers in code form can use a 'name' which must start with a letter, upper- or lower-case. You can add to that letter other letters, making a complete word if you like, or digits, but not spaces, arithmetic symbols (+, -, *, /) or punctuation marks, including the underscore (_). Names like TOTAL, lastname, ALLTHEREIS and R2D2 can all be used for number variables, and each can be assigned to a different number. One thing that you need to be careful about, though, is that the Spectrum 128 *does not distinguish between upper and lower-case names*. If you assign the variable name of jam to the number 45, and then assign the name of JAM to 88, you will find that both jam and JAM have been assigned to the same value. This will be whichever number was last to be assigned. Another thing to watch for is the use of *reserved words*. The reserved words of the Spectrum 128 are its instruction words, words like PRINT, NEW, RUN and so on. You *cannot* use these as variable names, and you will get an error indication, consisting of the cursor moving back over the word LET, if you attempt to use them. Some computers won't even allow you to use words that *contain* these reserved words, so that you could not use words like NEWLY. The Spectrum 128, however, is more tolerant, and will allow you to use any word which is not completely identical to a reserved word.

Just to make it even more useful, you can use variable 'names' to represent words and phrases, called 'strings' also, but there are some differences. One difference is that you have to add a dollar sign ($) to the variable name. The other difference is that the variable name for a string of characters can consist of *one letter only*, no more. If n is a variable name for a number, then n$ (pronounced en-string or en-dollar) is a variable name for a word or phrase. The computer treats the two variable names, n and n$, as being entirely separate and different. They also have to be assigned in rather different ways. When you assign a number to a number variable, using LET, you don't have to type a quote mark (") on each side of the number. When you assign a string variable in this way, however, you have to make use of quote marks. We'll look at other methods of making assignments later.

Serenade for strings

Figure 3.2 illustrates *string variables*, meaning the use of variable names for words and phrases. Lines 10 and 20 carry out the assignment operations, and lines 30 to 50 show how these variable names can be used. Notice that you can mix a variable name, which doesn't need quotes around it, with ordinary text, which *must* be surrounded by quotes. You have to be careful when you mix these two, because it's easy to run words together. Note in lines 30 to 50 how spaces have been left between words, and how we can use the apostrophe to force a new line to be taken. When you are printing one variable after another, the space is created by typing quotes, then pressing the spacebar, then another quote mark, like " ". To leave a space between text in quotes and a variable name, you only need to press the spacebar at the point where you need the space. The semicolons are essential when you are joining up bits of text in this way. If you omit the semicolon at a join, you will get an error indication, with the

cursor moving back to the variable name – try it to see just what happens when you press ENTER on the line.

Figure 3.3 shows another example, this time using the variable names B$ and P$ for longer phrases. There wouldn't be much point in printing messages in this way if you wanted the message once only, but when you continually use a phrase in a program, this is one method of programming it so that you don't have to keep typing it! The apostrophe mark has been used again in line 40 to force the printing to go on to another line without repeating the PRINT instruction.

Because the name of a string variable is marked by the use of the $ sign, a variable like a$ is not confused with a number variable like a. We can, in fact, use both on the same program knowing that the computer at least will not be confused. Figure 3.4 illustrates that the difference is a bit more than skin deep, though. Lines 10 and 20 assign number variables a and b, and string variables a$ and b$. When these variables are printed, you can't tell the difference between a and a$ or between b and b$. The Spectrum 128, like other computers, is arranged to work with numbers only when they are typed as numbers, or in the form of number variables. When numbers are placed between quotes, or assigned as string variables, then they can't be used in the way that you would normally use numbers for adding, subtracting, multiplying and dividing and so on. The result is that the computer can carry out arithmetic on two number variables, but not on two strings that represent number variables. When you think about, it would make little sense to allow arithmetic actions on string variables. You can multiply "2" by "3", but you can't multiply "2 LABURNUM WAY" by "3 ACACIA AVENUE". The computer therefore refuses to carry out multiplication, division, addition, subtraction or any other arithmetic operation on strings. Attempting to program a forbidden operation will cause an error indication, with the cursor moving back in the usual way. Later on, we'll see that there are operations that we can carry out on strings that we can't carry out on numbers, and attempts to do these operations on numbers will also cause an error. The difference is an important one. The computer stores numbers in a way that is quite different from the way it stores strings. The different methods are intended to make the use of arithmetic simple for number variables (for the computer, that is), and to make other operations simple for strings. Let's face it, it's only a machine!

On the Spectrum 128, the '+' sign will always cause addition of number variables for which addition is possible, but you will not get an error message if you try to add strings that do not consist of numbers. This is an operation of joining that can be carried out on strings, but not on numbers. It uses the + sign, but it isn't addition in the sense of adding numbers. Figure 3.5 illustrates this action of joining strings,

```
  10 LET a$="Sienna"
  20 LET b$="Brown"
  30 CLS
  40 PRINT "Just call me ";a$+"-
"+b$;", he"'" said."
  50 PRINT : LET a$="123": LET b
$="456"
  60 PRINT "Joined string is ";a
$+b$
```

Figure 3.5 Concatenating or joining strings. This is not the same action as addition!

which is called *concatenation*. This is nothing like the action of arithmetic, as you'll see by lines 40 and 60. Line 60 uses numbers in place of the names placed between the quotes. Concatenation is a very useful way of obtaining strings which otherwise would need rather a lot of typing. Take a look at Figure 3.6. This defines strings a$ and b$ as characters which can be used as 'frames' around a title. The title is defined in line 20 as The new Spectrum 128. Line 40 then prints a concatenated string that surrounds the title with asterisks and hash marks.

```
  10 LET a$="***": LET b$="###"
  20 LET s$="The new Spectrum 12
8"
  30 CLS
  40 PRINT a$+b$+s$+b$+a$
```

Figure 3.6 Using concatenation to make a frame for a title.

Getting some input

So far, everything that has been printed on the screen by a program has had to be placed in the program before it is run. We don't have to be stuck with restrictions like this, however, because the computer allows us another way of putting information, number or name, into a program while it is actually running. A step of this type is called an INPUT and the BASIC instruction word that is used to cause this to happen is also INPUT.

Figure 3.7 illustrates this with a program that prints your name. Now I don't know your name, so I can't put it into the program beforehand. What happens when you run this is that the words:

```
What is your name
```

are printed on the screen. On the bottom line of the screen, you will see a flashing letter "L" between quote marks. The computer is now waiting for you to type something, and then press ENTER. Until the ENTER key is pressed, the program will hang up at line 30, waiting for you. Whatever you type now appears between the quotes in the bottom line of the screen, and though this is a string, you don't need to type any more quotes – they are already waiting there. If you're honest, you will type your own name and then press ENTER. When you press ENTER, your name is assigned to the variable n$. The program can then continue, so that line 40 clears the screen and spaces down by two lines. Line 50 then prints the famous phrase with

```
  10 CLS
  20 PRINT "What is your name"
  30 INPUT n$
  40 CLS : PRINT : PRINT
  50 PRINT n$;" -this is your li
fe"
```

Figure 3.7 Using the INPUT instruction. The name that you type is put into the phrase in line 50.

```
10 CLS : PRINT "Enter a number
, please"
20 INPUT n
30 CLS : PRINT
40 PRINT "Twice ";n;" is ";2*n
```

Figure 3.8 An INPUT to a number variable. The quantity that you type must be a number.

your name at the start. You could, of course, have answered `Mickey Mouse` or `Donald Duck` or anything else that you pleased. The computer has no way of knowing that either of these is not your true name. Don't listen to the nut-cases who tell you that computers know everything!

We aren't confined to using string variables along with INPUT. Figure 3.8 illustrates an INPUT step which uses a number variable `n`. The same procedure is used. When the program hangs up with the "L" cursor appearing on the bottom line, you can type a number and then press the ENTER key. The action of pressing ENTER will assign your number to N, and allow the program to continue. Line 40 then proves that the program is dealing with the number that you entered. When you use a number variable in an INPUT step, then what you have typed when you press ENTER must be a number. If you attempt to enter a string, the computer will refuse to accept it, and you will get an error message – `2 Variable not found`, meaning that what you entered was not suitable for a number variable name. If your INPUT step uses a string variable, then *anything* that you type will be accepted when you press ENTER, but you will get an error message if you try to INPUT into a number variable on something that is not a number.

The way in which INPUT can be placed in programs can be used to make it look as if the computer is paying some attention to what you type. Figure 3.9 shows an example – but with INPUT used in a different way. This time, there is a phrase following the INPUT instruction. The phrase is placed between quotes, and is followed by a semicolon and then the variable name n$. This line 20 has the same effect as the two lines:

```
15 PRINT "Type your name, please";
20 INPUT n$
```

but with the difference that when the flashing `"L"` cursor appears on the bottom line, *the question appears with it*, and your reply is also on the same line – unless the length of the name causes letters to spill over on to the next line. You can also use a comma or an apostrophe in place of the semicolon in a line like this. Try the effect for yourself. The comma causes the name that you enter to be placed at the start of the next screen column, and the apostrophe causes the flashing `"L"` to be on the line under the question phrase. In this case, because the question extends into the second

```
10 CLS
20 INPUT "Type your name,pleas
e ";n$
30 PRINT
40 PRINT "Very pleased to meet
you, ";n$
```

Figure 3.9 Using INPUT to print a phrase which requests the input.

```
10 CLS
20 INPUT "Name and number, ple
ase ";n$,n
30 PRINT : PRINT
40 PRINT "The name is ";n$
50 PRINT "The number is ";n
```

Figure 3.10 Putting in two variables in one INPUT step. You need to be careful about how you enter the items, however.

column, the effect of the comma is to put the answer on the next line, but if the question is short, its effect is to put the answer into the second screen column.

The use of INPUT isn't confined to a single name or number. We can use INPUT with two or more variables, and we can mix variable types in one INPUT line. Figure 3.10, for example, shows two variables being used after one INPUT. One of the variables is a string variable n$, the other is the number variable n. Now when the computer comes to line 20, it will print the message at the bottom of the screen, and then wait for you to enter both of these quantities, a name and then a number. There is just one way of doing this correctly. That is to type the name, and then press ENTER. The computer will then shift the cursor to the next line or next available column. This means 'more needed', and that's a signal for you to type the number and then press ENTER again. The name and number will then be printed again in lines 40 and 50. If you use a semicolon between n$ and n, instead of a comma, then both the name and the number can be entered in the same line. *Note:* if you type the name, then a comma, and then a number and press ENTER, your machine will probably lock up – test this. Locking up means that no key has any effect, and you will have to press the reset button, and so lose your program. Be warned!

We can extend this principle further. Figure 3.11 calls for four numbers to be entered. These, once again, must be entered one by one, pressing ENTER each time. Once again, the numbers are assigned to the variable names, and the program will print the sum in line 40. Remember that you can put the message for your input higher up the screen by using PRINT, then INPUT. If you include the message with INPUT, then it always appears at the bottom of the screen.

```
10 CLS
20 INPUT "Four numbers, please
";a,b,c,d
30 PRINT
40 PRINT "The sum of these is
";a+b+c+d
```

Figure 3.11 An INPUT step which calls for four numbers which are then added.

Reading the data

There's yet another way of getting data into a program while it is running. This one involves reading items from a list that is part of the program, and it uses two instruction words READ and DATA. The word READ causes the program to select an item from the list. The list is marked by starting each line of the list with the word DATA. The items of the list can be separated by commas. Each time an item is read from such a list, a 'pointer' is altered so that the next time an item is needed, it will be the next item on the list rather than the one that was read the last time round.

We'll look at this in more detail in Chapter 5, but for the moment we can introduce ourselves to the READ . . DATA instructions. Figure 3.12 uses the instructions in a very simple way. Line 20 reads an item number, which is the first item on the list and assigns it to the variable `j`. This is printed in line 30, with the semicolon keeping printing in the same line so that the phrase in line 40 follows it. The semicolon at the end of line 40 once more keeps the printing in the same line, and line 50 reads the name which is the second item in the list. This is assigned to the variable name `n$` and printed in line 60. Line 70 contains the DATA, one number and one phrase. The number can be typed as it is, but you can see that the phrase needs quotes round it. Anything that you assign to a string variable in this way must have quotes around it in its DATA line. If you omit the quotes, you'll get an error message (in this example) of `Variable not found 50:1`, meaning that something went wrong in line 50. Not, you'll notice in line 70 where the real fault is. The machine can only tell that something did not fit the variable name `n$` in line 50, it can't tell where this item came from. In addition, you always have to be careful about how you match your READ and your DATA. If you use a number variable in the READ, like `READ a`, then what is in the DATA line being read *must* be a number. If you use a string variable, as in `READ a$`, then it doesn't matter whether your DATA line contains a number or a string, but it must be placed in quotes. Remember that if you read a number using `READ a$`, then the Spectrum 128 does not allow you to carry out any arithmetic on that number.

Now for an odd experience. If you have typed the program of Figure 3.12, and run it, you will have seen it operate. Now type GO TO 20 , and press ENTER. This will make the program start at line 20. This time, instead of the program running, you get an error message:

```
Out of data 20:1
```

```
10 CLS
20 READ j
30 PRINT " Item ";j;
40 PRINT " is a ";
50 READ n$
60 PRINT n$
70 DATA 5,"disk drive"
```

Figure 3.12 Using the READ and DATA words to place information into a program. This is used only for fixed items that are always part of the program.

What does this mean? Quite simply, it means that you have read both of the items in the data line, and there aren't any more! The Spectrum 128 does not start its DATA list all over again each time you use GO TO, though it does when you use RUN. There is a way round this, which is to type the single line instruction:

```
RESTORE
```

before you type `GO TO 20`. RESTORE means 'go to the start of the DATA list again', and it's an instruction that we'll be looking at again, because it has other uses. Meanwhile, though, you should remember that any program which makes use of READ . . DATA might need a RESTORE if you want to read the data again in the same program. The READ . . DATA instructions really come into their own when you have a long list of items that are read by repeating a READ step. These would be items that you would need every time that the program was used, rather than the items you would type in as replies. We're not quite ready for that yet, so having introduced the idea, we'll leave it for now.

Number antics

The amount of computing that we have done so far should have persuaded you that computers aren't just about numbers. For some applications, though, the ability to handle numbers is very important. If you want to use your computer to solve scientific or engineering problems, for example, then its ability to handle numbers will be very much more important than if you bought it for games, for word processing or even for accounts. Don't be fooled by the size or price of the Spectrum, it's just as capable of dealing with scientific or engineering problems as larger machines. It's time, then, to take a very brief look at the number abilities of the Spectrum 128. It *is* a brief look because we simply don't have space to explain what all the mathematical operations do. In general if you understand what a mathematical term like *Sin* or *Tan* or *Exp* means, then you will have no problems about using these mathematical functions in your programs. If you don't know what these terms mean, then you can simply ignore the parts of this section that mention them.

The simplest and most fundamental number action is *counting*. Counting involves the ideas of *incrementing* if you are counting up and *decrementing* if you are counting down. Incrementing a number means adding 1 to it, decrementing means subtracting 1 from it. These actions are programmed in a rather confusing-looking way in BASIC, as Figure 3.13 shows. Line 20 sets the value of variable `x` as 5. This is printed in line 30, but then line 40 increments `x`. This is done using the odd-looking

```
  10 CLS
  20 LET x=5
  30 PRINT "Value of x is ";x
  40 LET x=x+1: PRINT
  50 PRINT "Now we've used LET x
=x+1": PRINT
  60 PRINT "The value of x is ";
x
```

Figure 3.13 Incrementing, using the equals sign to mean 'becomes'.

```
10 CLS
20 LET x=5: PRINT "x is ";x
30 PRINT
40 LET x=2*x+4
50 PRINT "It's changed _ "
60 PRINT "x is now ";x
```

Figure 3.14 A more elaborate assignment, using an *expression*.

instruction: `LET x = x + 1`, meaning that the new value that is assigned to `x` is 1 more than its previous value. The rest of the program proves that this action of incrementing the value of `x` has been carried out.

The use of the `=` sign to mean 'becomes' is something that you have to get accustomed to. When the same variable name is used on each side of the equality sign, this is the use that we are making of it. We could have a line:

```
LET x = x-1
```

and this would have the effect of making the new value of `x` one less then the old value. Variable `x` has in this instance been *decremented* this time. We could also use `LET x = 2*x` to produce a new value of `x` equal to double the old value, or `LET x = x/3` to produce a new value of `x` equal to the old value divided by three. Figure 3.14 shows another assignment of this type, in which both a multiplication and an addition are used to change the value of `x`.

Take precedence

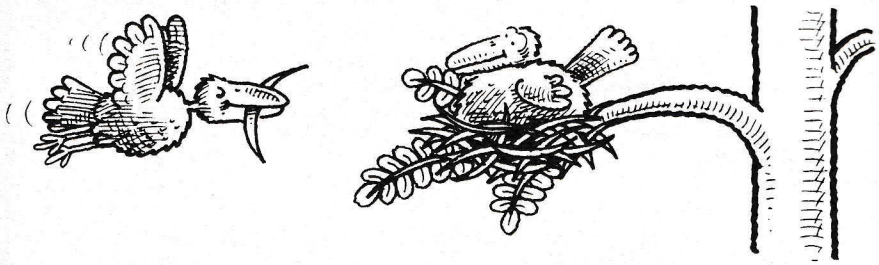
You'll have noticed that line 40 of Figure 3.14 used `LET x=2*x+4`. Now this consists of two operations, a multiplication and an addition. The question is, what do we expect to get from this when `x` is 5? Do we get 2 times 5 equals 10, then 10 plus 4 is 14, or do we get `5` plus 4 is 9, and 2 times 9 is 18? As it happens, we get the result fourteen, because the multiplication is always carried out before the addition. This isn't just because it comes first in the line, either. If you changed the line to `LET x=4+2*x` you would still get the same result. The reason is that these operations are done in a fixed order, called the *order of precedence*. This is listed in Figure 3.15, and

Order of priority

For ordinary arithmetic, order of priority is MDAS – multiplication and division, followed by addition and subtraction. The full order of priority is:

1. **Raising to a power, using ^**
2. **Multiplication and division**
3. **Addition and subtraction**
4. **Comparison, using = < >**
5. **AND**
6. **OR**
7. **NOT**

Figure 3.15 The order of precedence for simple arithmetic actions.



you can see that the operation of raising to a power is the one that comes first in any line that contains several number operations. There are some actions that have even higher priority, but these will keep until later. To illustrate the precedence laws, if you have something like:

```
PRINT 5-3*6^2
```

– then you'll get the answer -1Ø3. That's because 6^2 (six squared) is done first, giving 36. Multiplying by 3 gives 108, and taking 108 from 5 gives –103. A set of operations in one line like this is called an *expression*, especially when one or more of the numbers is represented by a variable name. In any expression, then, the actions are carried out in order of precedence. If you end up with a set of actions that are all of the same precedence, like a set of additions and subtractions, then the order is left-to-right, just as you read the line.

There's one big exception to the rules, though. If you put any part of an expression within brackets, then it gets priority over everything else. For example, if you had the line:

```
PRINT 3+(5-2)^3
```

then you get the answer 3Ø. This is because 5–2=3 is done first. The raising to the power three (cubing) then gives 3^3=27, and the addition comes last, giving 30. If you are uncertain about what might happen in an expression, then you can always force the machine to obey your own order by putting inside brackets whatever you want done first. You can even have brackets contained inside other brackets, a system called 'nested brackets'. When you do this, whatever is inside the *innermost* brackets gets done first, then whatever is inside the outer brackets, and so on. You will often need to use brackets if your program calls for an equation to be worked out. You'll be surprised how often this kind of thing is needed, even just for screen displays.

```
  10 CLS : LET x=2.5
  20 PRINT "It'square root is ";
SQR (x)
  30 PRINT
  40 PRINT "It's natural log. is
 ";LN (x)
  50 PRINT
  60 PRINT "It's ordinary log is
 ";LN (x)/2.303
```

Figure 3.16 Using some number functions. Note that the only form of logarithm that is provided is the natural log, LN. The program shows how to convert to common logs, as used in engineering work.

Note: **The number functions require a number, called the *argument* following them. Where this is a number or a single number-variable, you can usually either enclose it in brackets or separate it from the instruction word by one space. If the argument is an expression, such as:**

```
3*x-y
```

then it *must* be enclosed in brackets. Functions that work with angles all use the angle value in *radians*. To convert radians to degrees, multiply by 180/PI. To convert degrees to radians, multiply by PI/180.

ABS (X)	**Converts negative sign to positive.**
ACS (X)	**Gives angle in radians whose cosine is** X.
ATN (X)	**Gives angle (in radians) whose tangent is** X.
BIN X	**Converts binary number** X **into an ordinary (denary) number.**
COS (X)	**Gives the cosine of angle** X **(radians).**
EXP (X)	**Gives the value of e to the power** X.
INT (X)	**Gives the whole-number part of** X**, rounded to the nearest *smaller* whole number.**
LN (X)	**Gives the natural logarithm of** X.
PI	**Gives the value of pi, the ratio of the circumference to the diameter of a circle.**
RANDOMIZE (X)	**Sets new sequence of 'random' numbers. Best used with no argument.**
RND	**Gives a random fraction between 0 and 1.**
SGN (X)	**Gives the sign of** X**. The result is +1 if** X **is positive, -1 if** X **is negative, 0 if** X **is zero.**
SIN (X)	**Gives the sine of angle** X **(radians).**
SQR (X)	**Gives the square root of** X**. The argument** X **must be a positive number.**
TAN (X)	**Gives the value of the tangent of angle** X **(radians).**

Figure 3.17 Number functions, with brief notes. Don't worry if you don't know what some of these do. If you don't know, you probably don't need them!

```
10 CLS : LET x=2.5
20 PRINT 1/3,2/3
30 PRINT 1/11,10/11
40 PRINT 1/3+2/3,1/11+10/11
```

Figure 3.18 How the computer copes with 'awkward' numbers. Very small or very large numbers are converted into *standard form*.

Number functions

Figure 3.16 illustrates the use of a few number functions. A number function in this sense is an instruction which operates on a number to produce another number. Line 10 picks the value of 2.5 for `x`. Line 20 then prints the value of the square root of the number that has been assigned to `x`, for which we use the instruction word SQR. An alternative is to use the `^` operation in the form `x^.5`, but `SQR(x)` is easier to type and remember. For other roots, like the cube root, you can use expressions like `x^(1/3)` and so on. `LN(x)` then produces the natural logarithm of `x`. This is not the type of logarithm that you may want, and to find the ordinary (base 10) log, you have to use `ln(x)/2.3Ø3`.

Figure 3.17 illustrates the various number functions that can be used, with a brief explanation of what each one does. Some of these actions will be of use only if you are interested in programming for scientific, technical or statistical purposes. Others, however, are useful in unexpected places, such as in graphics programs.

How precise?

One of the problems of small computers is precision of numbers. You probably know that the fraction 1/3 cannot be expressed exactly as a decimal. How near we can get to its true value depends on the number of decimal places we are prepared to print, so that 0.33 is closer than 0.3, and 0.333 is closer still. The computer converts most of the numbers that it works with into the form of a fraction and a multiplier. The fraction is not a decimal fraction but a special form called a binary fraction, and this conversion is seldom exact. The conversion is particularly awkward for numbers like 1, 10, 100 and also .1, .01, .001; all the powers of ten, in fact. To avoid embarrassments like printing `3-2 = .9999999`, the computer will round numbers of this type up or down as need be before displaying them. Not all computers do this well – you can be glad that you bought a Spectrum 128! Figure 3.18 shows how the Spectrum 128 copes with fractions like 1/3, 2/3, 1/11 and 10/11. The numbers that you see on the screen have been rounded to seven places of decimals, but the number that the computer *stores* must be of many more decimal places. The rounding works to make sure that adding the fractions gives the correct result.

Another point that belongs here is that there is an alternative method of entering numbers. If you have worked in any subject that uses very large or very small numbers (Physics, Chemistry, Engineering), you will know about this method already. If you haven't met it before, it's called Standard Form. A number in standard form consists of a quantity which lies somewhere between 0 and 10, never quite equal to either of these, and multiplied by a power of ten. Take, for example, a number like 132 000. If we shift the decimal point five places *left* (equivalent to *dividing* by ten to the power 5), then this becomes 1.32. To get the value correct, this would have to be multiplied by 10 to the power 5, or, as we write it E5. The number 132 000 could therefore be written as 1.32E5.

Suppose we try a small number, 0.00036. This is 3.6 times ten to the minus 4, or 3.6E–4, with the minus sign there because we have had to shift the decimal point four places *right* to get this result. The Spectrum 128, like most small computers, will accept and print numbers in this form. The conversion is automatic, and the

```
10 CLS : LET x=2.5
20 PRINT "Enter three numbers,
please "
30 INPUT a,b,c
40 PRINT "Sum is ";FN s(a,b,c)
50 DEF FN s(x,y,z)=x+y+z
```

Figure 3.19 Using a very simple defined function.

Spectrum 128 will display numbers in this form only when it has to – which means when the numbers would need more than eight figures to display. You can, however, enter numbers such as `1.6E4` if you want to, and this will be printed as `16000`. When you have a line such as `INPUT a`, using a number variable, then the letter `e` will be accepted in a number like `1.6e5`, but no other letters will be accepted.

Another action which is specially useful both in mathematical work and for strings is defined functions. This is a way of making use of a mathematical action many times, but writing it just once into your program. Figure 3.19 shows a example of using a defined function. It's a very simple example, and you would never use a defined function for anything so easy, but being easy makes it simpler to follow. Lines 20 and 30 ask you to input three numbers. Line 40 then prints a quantity called '`FN s(a,b,c)`'. Now this *function* has no meaning unless it is defined, and it has to be defined by a line that starts with DEF FN. When you enter this line, you have to follow this with the name that you have decided on which in this example is `s`. This has to be a single-letter name. The same line then shows what the function has to do, add the three numbers. Now you'll notice that the functions has been defined here with variable names `x`, `y` and `z`, not `a`, `b` and `c`. This is one of the useful features of a defined function – you don't have to use the same letter-names in it as you use in the main program. What follows the equals sign in this line then describes what has to be done with the three numbers. This again, has been written in terms of `x`, `y` and `z`. When the function is *called* by using `FN s(a,b,c)`, then whatever you assigned to `a` in your program gets passed to `x` in the function, and similarly whatever was assigned to `b` is transferred to `y`, and what was assigned to `c` is transferred to `z`. In the defined function, these are added, and the result is what appears when you print. You could also have made a new assignment, such as:

```
LET k=FN s(a,b,c)
```

– so that the variable `k` took the value of `a+b+c`. Obviously in this case it would be easier just to use `LET k=a+b+c`, but it's easier to see what is going on in a simple example.

The really useful part of a defined function is that it can use variable names of its own. The names must match up, because if you want the function to do `a+b-c`, then you have to type `x+y-z` in the function and make sure that `x`, `y`, `z` are in the same order in their brackets as `a`, `b`, `c`. In other words, you can't have `FN s(a,b,c)` along with `DEF FN s(y,z,x)=x+y-z` when you want to get `a+b-c`. Later on, we'll look at string defined functions that allow you to work with both strings and numbers.

Figure 3.20 shows another example, showing this time that you can put the DEF FN part at the start of a program just as easily as at the end. In this case, the definition is the square root of the sum of the squares of two numbers, and this will be the answer that is returned to the main program in line 40. Once again, the letters that are used

```
10 DEF FN h(a,b)=SQR (a^2+b^2)
20 CLS
30 INPUT "Two triangle sides,
please ";x,y
40 PRINT "The third is ";FN h(
x,y)
```

Figure 3.20 Another defined function, this time for working out the length of the longest side of a right-angled triangle.

in the DEF FN part are not the same as are used in the FN part, because the values are passed on to them. This allows you to use this DEF FN with any pair of letters, or even with numbers. You can, for example, after this program has run, type:

```
print fn h(5,12)
```

– and get the answer 13 directly. It's just as if you had added a new action to your Spectrum BASIC for finding the longest side of a right-angled triangle. In fact, you *have* added this new function, but unlike the others, it's available only after line 10 of this program has run.

Translating formulae

Perhaps you think that you'll never need to put a mathematical formula into BASIC form. Perhaps not, but there are astonishingly few things that don't need something of the sort, and you wouldn't thank me if I just left the subject out. A formula is just an expression that shows what is to be done with numbers. What makes $y = 3x + z$ a formula and $4 = 3*2-2$ not is that the formula is general, it uses variable names rather

Formula *Textbook form*	**Formula** *BASIC form*
$A = \pi r^2$	`LET A=PI*R^2`
$V = \frac{4}{3}\pi r^3$	`LET V=(H*PI*R^3)/3`
$N = ar^{n-1}$	`LET N=A*R^(N-1)`
$\text{Sin}(A+B) = \text{Sin A Cos B} + \text{Cos A Sin B}$	`LET X=SIN(A)*COS(B)+COS(A)*SIN(B)`
$\text{Cos } 2A = \text{Cos}^2 A - \text{Sin}^2 A$	`LET X=(COS(A))^2-(SIN(A))^2`
$a^2 = b^2 + c^2 - 2bc \text{ Cos A}$	`LET A=SQR(B^2+C^2-2*B*C*COS(A))`
$(so\ a = \sqrt{(b^2 - 2bc \text{ Cos A})}$	
$a = \frac{-b \pm \sqrt{b^2 - 4ac}}{2c}$	`LET X1=(-B+SQR(B^2-4*A*C))/2*A` `LET X2=(-B-SQR(B^2-4*A*C))/2*A`

Figure 3.21 A few examples of formulae re-worked so that they can be used in a BASIC program.

than just numbers. Because a variable name in BASIC can be a single letter, a lot of formulae don't really need much working on. For example, if you have the formula to find the area of a circle, which is written as $A = Pi \times r^2$, then this becomes in BASIC:

```
LET A=pi*r^2
```

– pretty much the same as the formula. This is because there's no problem about precedence, the square is done first, then multiplication by pi. If you didn't know that you could get *pi* on your keyboard, then just type: `print pi` and press ENTER, and you'll see the value 3.1415927 appear.

Formulae get more complicated when you need to use brackets to get the order right. You really have to know what the formula is about before you can get it correctly translated into BASIC, and for that reason, I won't give a long string of examples, apart from the few in Figure 3.21. The point here is that if you need to use these kinds of formulae, you probably know what they are for, and it will be straightforward to translate them. If you don't know what they are for, then you'll need a textbook of maths, not of computing! This will be about the last time that mathematics rears its head in this book, because we're going on now to look at what can be done with strings, which means computing with text and graphics. One last point, though. If you want at any stage to erase all assignments to variables, you can use the word CLEAR. This is done automatically when you use RUN, but you can use CLEAR at other stages in a program to clear out all old variable values if you need to.

Chapter 4

Repeating yourself

One of the activities for which a computer is particularly well suited is repeating a set of instructions, and every computer is well equipped with instructions that will cause repetition. The Spectrum 128 is no exception to this rule, and it is equipped with two very useful 'repeat' commands. We'll start with one of the simplest of these repeating actions, GO TO. Though the action is simple, and it's easy to use, GO TO causes more trouble than anything else in BASIC, for reasons that you'll appreciate later. In fact, it needn't, and if you learn to program correctly in BASIC, then GO TO need never be a problem. If it's any consolation, every computing language has GO TO in it in some form or another, so we needn't be worried about using it. GO TO means just what it says – go to the line whose number follows the word GO TO. If the line that you GO TO is later in the program, the effect of GO TO is to skip part of the program. If the line that you GO TO is earlier in the program, the effect is to repeat some lines, forming what we call a *loop*.

Figure 4.1 shows this sort of thing in action. Line 10 contains a print instruction, and line 20 is `GO TO 1Ø` which simply makes line 10 run again. This would run indefinitely if it were not a printing action, but since it does print, the phrases fill the screen, and then you get the message about scrolling. If you press the 'n' key, the program will break instead of going on endlessly. Now the trouble with GO TO, even in such a simple example as this, is that only the end of the loop is marked, because this is where the GO TO is. More advanced versions of BASIC have commands for loops in which both the start and the end are marked so that you can see at a glance where the loop begins and where it ends. As it happens, though, you *can* do this on the Spectrum 128, and it makes programming much easier, particularly when you are working with longer programs. For that reason, we'll illustrate this method in this book.

```
  10 PRINT "This is a Spectrum l
oop"
  20 GO TO 10
```

Figure 4.1 A *very* simple loop. You can stop this by pressing the BREAK key.

```
10 LET n=0
20 LET repeat1=20
30 LET n=n+1
40 PRINT n;" Spectrum printing
line"
50 GO TO repeat1
```

Figure 4.2 A loop which carries out a counting action very rapidly. You will also have to use the BREAK key to stop this one.

Figure 4.2 shows an another example of a very simple loop, using a 'marker' at the start. Line 20 contains the instruction, `LET repeat1=20`. This acts as a marker, identifying line 20 as the start of a loop – the word `repeat` is a good reminder. By making this assignment, `repeat1` means 20. When this has been done, the program moves on to line 30, which increments the number value of `n` that started at zero because of line 10. Line 40 then instructs the machine to print the phrase `Spectrum printing line` following the number. In this simple example, that's all we ask it to do. Line 50 then ensures the repeating action, using `GO TO repeat1`. If you try to omit the name, you will get an error message. Once again, there is nothing in this set of instructions which could cause the loop to stop repeating, and so it will cause the screen to fill with the words:

```
Spectrum printing line
```

until you press the 'n' key when you are asked about scrolling. Any other type of loop that appears to be running forever can normally be stopped by pressing the BREAK key, then ENTER. In addition, when this loop runs, you will see a different number printed out each time the computer goes through the actions of the loop. We call this 'each pass through the loop'.

Now a never-ending loop like this is not exactly good to have, and GO TO is a method of creating loops that we can use much more constructively. To do this, we need a way of stopping the loop when we want to. This will be when some condition has been fulfilled. You might, for example, want to stop a loop when a number variable reaches a value of 100, or when the name `LASTONE` is entered. The extra commands that you need to know about in order to do this are IF and THEN.

IF has to be followed by a condition. You might use conditions like `IF N=20`, or `IF N$="LASTONE"` for this purpose. After the condition, you can use the word THEN and follow this with another instruction, such as a GO TO. The use of GO TO is *not* optional, however, as it is on some machines, and you cannot use instructions like `THEN 10` or `THEN repeat1`. You can use an equality condition like `IF n$="lastone"` or an inequality condition like `IF n$<>"lastone"`, where the <> signs together mean 'not equal to'. As we'll see later, you can have several conditions between the IF part of the test and its end at the THEN GO TO part.

Now if all of that sounds rather complicated, take a look at the simple illustration in Figure 4.3. Lines 10 to 30 contain the instructions. You are allowed to enter names, and the program will stop repeating the entry process when you enter the name of `lastone`. How do we program this? We start in line 40 with the `LET|repeat1=40` instruction marking the start of this loop, using `repeat1` once again as the marker word. What we have to do over and over again is to INPUT a name, so line 50 attends to that, and the name is printed in line 60. A 'real' program would do something with the name, but this is just an example to get you used to the techniques. After the name has been typed and the ENTER key pressed, you have to test the name to find if it is `lastone`. This is done in line 70. The test is to find if `n$`

```
  10 CLS
  20 PRINT "Enter some names - "
  30 PRINT "- use lastone to sto
p loop"
  40 LET repeat1=40
  50 INPUT n$
  60 PRINT n$
  70 IF n$<>"lastone" THEN GO TO
 repeat1
```

Figure 4.3 Using a test condition to stop a loop from running. Line 70 contains the test, and if the test result is false, the program ends.

is *not* lastone, because in that case we want to go back and do the entry and print steps again. What do we do if it lastone? The answer is to stop, since there are no more lines of program to carry out. If, however, we had a line 80, this line would then be carried out after lastone was entered.

Now when you run this, you'll find that it ends the loop when you type lastone, but not for Lastone or LASTONE. This is quite easy to deal with, because you can add some AND's to your test. For example, if you alter line 70 so that it reads:

```
70 IF n$<>"lastone" AND n$<>"Lastone" AND n$<>"lastone"
   THEN GO TO repeat1 ???
```

– you are making three conditions in place of one. If *any one* of these conditions is true, the GO TO repeat1 will not be carried out and the loop will stop. You can also use the word OR to couple two conditions, and we'll look at examples of this type of thing later. As an example of more complicated sets of conditions, take a look at Figure 4.4.

```
  10 CLS
  20 PRINT "Heads or Tails"
  30 PRINT "Type E to stop"
  40 LET repeat1=40: LET exit=11
0: LET k=0
  50 LET x=INT (RND*2)+1
  60 INPUT "Heads(h)or Tails(t)?
";s$
  70 IF s$="e" OR s$="E" THEN GO
 TO exit
  80 IF s$="h" AND x=1 OR s$="t"
 AND x=2 THEN PRINT "You win": L
ET k=1
  90 IF k=0 THEN PRINT "You lose
!": LET k=0
 100 GO TO repeat1
 110 PRINT "Game over"
```

Figure 4.4 More complicated tests being used on a loop, in this case to form a simple heads-or-tails game.

RND by itself will give a number that will be between 0 and 1. It will never be exactly equal to 0 or 1. Suppose, then, that `RND` **gives a very small number, like .001. Multiplying by 2 will give .002, and the** `INT` **of this is 0, so that** `1+(RND*2)=1`**. If, however,** `RND` **gives a number like .9, then** `RND*2=1.8`**, and the** `INT` **of this is 1. Adding 1 then gives 2. In this way, then,** `1+INT(RND*2)` **will give either 1 or 2, depending on whether** `RND` **gives a fraction greater than .5 or less than .5.**

Figure 4.5 Illustrating how RND is used to give the numbers 1 or 2 only.

This is a simple heads-or-tails gamble, with no scoring. Lines 10 to 30 set things up as usual, while line 40 starts the main loop of repeated actions. Line 50 is the important gambling line. RND means 'select at random', and it select a number that is always a fraction, always less than 1. By multiplying this number RND by 2, we get a number that must be more than zero, but less than two. When we add 1 to this, we get a number that can be just over 1 or just less than 3, and by using INT (which removes fractions) we will get a number that is either 1 or 2. Figure 4.5 shows how this system works in this instance. Each time line 50 runs, then, either a 1 or a 2 will be assigned to variable name `x`. Line 60 then asks you to type `H` or `T` for heads or tails, and you have been told earlier that typing `e` will get you out of the game. Whatever you type is assigned to a variable called `s$`. Now we have to test `s$`. The first test is to find if we want to break out of the loop. This is done by typing `e` (on its edge!), and line 70 makes this test. We use the name `exit` here because we want the program to stop if `e` is typed – the name has already been assigned. The other tests are made in line 80. If we take X = 1 as meaning 'heads' and X = 2 as meaning 'tails', then this line detects success. The line will then print `You win`, and the number `k` is 1, making the program skip the action in line 90. The loop then repeats because of the GO TO in line 100. Line 90, however, checks failure. If your guess, H or T, did not agree with the selection of `x`, the value of `k` is zero, and the action of this line operates. This prints `You lose`, makes `k` equal to zero, and once again, allows the loop to continue. Only if the `E` has been typed does the game end with the message in line 110.

The FOR . . NEXT loop

There is an alternative to the GO TO type of loop when you want actions to be repeated a number of times. I really do mean *number*, in the sense that the condition that ends the loop is not a string being equal to another string, but a number count finishing. As the name suggests, this makes use of two new instructions, FOR and NEXT, that mark the start and the end, respectively, of the loop. The instructions that are repeated are the instructions that are placed between FOR and NEXT. The loop needs to make use of a number variable, and the name of this variable *must* follow both the FOR and the NEXT instruction words.

Figure 4.6 illustrates a very simple example of this type of FOR loop in action. The line which contains FOR must also include a number variable which is used for counting, and numbers which control the start of the count and its end. In the example, `n` is the counter variable, and its limit numbers are 1 and 10. The action is a

```
10 CLS
20 FOR n=1 TO 10
30 PRINT "Spectrum rules OK?"
40 NEXT n
```

Figure 4.6 Using the FOR..NEXT loop for a counted number of repetitions.

PRINT, and there is nothing else to do. The `NEXT n` in line 40 marks the end of the loop, and at this point, the machine will go back to the FOR line, increase the value of variable `n` by 1, and test to find if the end value of 10 (in this example) has been exceeded. If it has not, the loop runs again. If `n` is more than 10, then the program jumps to the line following the `NEXT n` line. Since there isn't one in this example, the program ends. The effect of this program, then, will be to print `Spectrum rules OK?` ten times. At the first pass through the loop, the value of `n` is set to 1, and the phrase is printed. When the end of the line is encountered, the computer increments the value of `n`, from 1 to 2 in this case. It then checks to see if this value exceeds the limit of 10 that has been set. If it doesn't, then the PRINT instruction is repeated, and this will continue until the value of `n` exceeds (not equals) 10 – we'll look at that point later. The effect in this example is to cause ten repetitions.

This, however, is a very simple sort of loop, and we normally use loops that contain several lines of instructions between the start and the end. You don't have to confine this loop action to single loops either. Figure 4.7 shows an example of what we call *nested loops*, meaning that one loop is contained completely inside another one. When loops are nested in this way, we can describe the loops as inner and outer. The outer loop starts in line 20, using variable `n` which goes from 1 to 10 in value. Line 30 is part of this outer loop, printing the value that the counter variable `n` has reached. Lines 40 and 50, however, contain another complete loop. This must make use of a different variable name, and it must start and finish again before the end of the outer loop. We have used variable `j`, and we have put nothing between the FOR part and the `NEXT j` part to be carried out. All that this loop does, then, is to waste time, making sure that there is some measurable time between the actions in the main loop. The last action of the main loop consists of clearing the screen following the `NEXT j` in line 50. The overall effect, then, is to show a count-up on the screen, slowly enough for you to see the changes, and wiping the screen clear each time. When you use nested loops of this kind, you must be absolutely sure that you have put the correct variable names following each NEXT. If you use a variable name (for example, `k`) that you haven't assigned, you will get message like '2 Variable not found, 50:1' showing that an incorrect variable name has been found in line 50, first part. If you mix up the variables, and make line 50 contain `NEXT n`, then the count will be rather fast!

```
10 CLS
20 FOR n=1 TO 10
30 PRINT "Count is ";n
40 FOR j=1 TO 200
50 NEXT j: CLS
60 NEXT n
70 PRINT "End of count"
```

Figure 4.7 A program that uses nested loops, with one loop inside another. The 'inner' loop here, in lines 40 and 50, acts as a time delay.

```
10 CLS
20 FOR n=10 TO 1 STEP -1
30 PRINT n;" seconds and count
ing."
40 FOR j=1 TO 200
50 NEXT j: CLS
60 NEXT n
70 PRINT "Blastoff!"
```

Figure 4.8 A countdown program, making use of STEP to decrement the number in each loop.

Even at this stage it's possible to see how useful this FOR . . NEXT loop can be, but there's more to come. To start with, the loops that we have looked at so far count upwards, incrementing the number variable. We don't always want this, and we can add the instruction word STEP to the end of the FOR line to alter this change of variable value. We could, for example, use a line like:

```
FOR n=1 TO 9 STEP 2
```

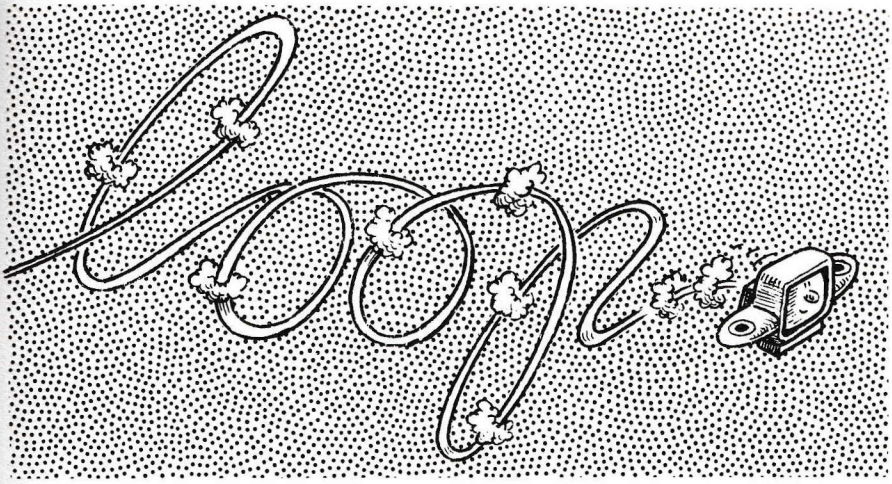
which would cause the values of n to change in the sequence 1, 3, 5, 7, 9. When we don't type STEP, the loop will always use increments of +1.

Figure 4.8 illustrates an outer loop which has a step of –1, so that the count is downwards. Variable n starts with a value of 10 this time, and is *decremented* on each pass though the loop. Line 40 once again forms a time delay so that the countdown takes place at a civilised speed. Remember that you can't omit the j following the next in line 50. A delay of this type is a particularly useful way of slowing a count down if a PAUSE can't be adjusted to exactly the tie that is needed but there's no such simple way to speed up a loop!

End values

Every now and again, when we are using loops, we find that we need to use the value of a counter variable such as n after the loop has finished. It's important to know what this will be, however, and Figure 4.9 brings it home. This contains two loops, one counting up, the other counting down. At the end of each loop, the value of the counter variable is printed. This reveals that the value of n is 6 in line 50, after completing the FOR n = 1 TO 5 loop, and is 0 in line 90 after completing the FOR n = 5 TO 1 STEP -1 loop. If you want to make use of the value of n, or whatever variable name you have selected to use, after a loop has finished, you will have to remember that it will have been incremented or decremented one more time beyond the final value that is specified in the loop command.

One of the most valuable features of the FOR . . NEXT loop, however, is the way in which it can be used with number *variables* as well as just numbers. Figure 4.10 illustrates this in a simple way. The letters a, b and c are assigned as numbers in the usual way in line 20, but they are then used in a FOR . . NEXT loop in line 30. The limits are set by a and b, and the step is obtained from an expression, b/c. The rule is that if you have anything that represents a number or can be worked out to give a



number, then you can use it in a loop like this, and this is generally true in other types of instructions as well.

```
10 CLS
20 FOR n=1 TO 5
30 PRINT n
40 NEXT n
50 PRINT "n is now ";n
60 FOR n=5 TO 1 STEP -1
70 PRINT n
80 NEXT n
90 PRINT "n is now ";n
```

Figure 4.9 Finding the value of the loop variable after a loop action is completed. This is always one more step than the specified end of the loop.

```
10 CLS
20 LET a=2: LET b=5: LET c=10
30 FOR n=a TO b STEP b/c
40 PRINT n
50 NEXT n
```

Figure 4.10 A loop instruction that is formed with number variables.

Loops and decisions

It's time to see loops being used rather than just demonstrated. A simple application is in totaling numbers. The action that we want is that we enter numbers and the computer keeps a running total, adding each number to the total of the numbers so far. Now if we used a FOR . . NEXT type of loop here, we would only be able to enter a fixed amount of numbers, which means that we would have to count how many numbers were to be totalled each time. That's too much like hard work, and it would be a lot more convenient if we could just stop the action by signaling to the computer in some way, perhaps by entering a value like 0 or 999. A value like this is called a *terminator*, something that is obviously not one of the normal entries that we would use, but just a signal. For a number-totaling program, a terminator of 0 is very convenient, because if it gets added to the total it won't make any difference.

Figure 4.11, therefore, shows an example of this type of program in action. We can't use a FOR . . NEXT type of loop, because we don't know in advance how many times we might want to go through the loop, so we have to use the GO TO type of loop, with the word `repeat` used to mark the start. The instructions appear first, and we then have to make the total variable `total` equal to zero in line 40. When a program is RUN, each variable like this is automatically set to zero in any case, but it's good practice to set it in the program. The reason is that if you start the program in any other way than RUN (like GO TO 10), the variables are *not* automatically set to zero. You might think that you would never do anything like that, but that's what they all say, your honour. Line 40 is the start of the main loop, using `repeat` loop, and each time you type a number, in response to the request in line 50, the number that you type is assigned to variable `n`. This value of `n` is then tested in line 60. If the value is 0, then the GO TO `exit` command takes effect, and the next line that will be carried out will be line 100, since the number 100 was assigned to `exit`. If the value of `n` was not zero, it is added to the total in line 70, and line 80 prints the value of the total so far. Line 90 contains the GO TO `repeat` command, so this is the end of the loop.

```
  10 CLS : PRINT AT 4,10;"TOTAL
FINDER"
  20 PRINT AT 6,1;"The program w
ill total numbers"'"for you. Ent
er 0 to stop."
  30 LET total=0
  40 LET repeat=40: LET exit=100
  50 INPUT "Number, please ";n
  60 IF n=0 THEN GO TO exit
  70 LET total=total + n
  80 PRINT "Total so far is ";to
tal
  90 GO TO repeat
 100 PRINT "Final total is ";tot
al
```

Figure 4.11 A running-total program which can't use FOR..NEXT. Line 60 carries out the test which decides whether or not to loop.

The effect, then, is that if the number which you have typed in line 50 was not a zero, line 90 will send the program back to repeat from line 50 onwards. This will continue until you do enter a zero. When this happens, the test in line 60 succeeds (n is zero), and the program jumps to line 100, leaving the loop. This line prints the final value of 'total', and the program ends. As always, when you have a program that contains an INPUT to a number variable like n, you have to be sure that you enter a number. If you try to enter a letter, you'll see the error message `2 Variable not found, 5Ø:1` appear, because there is no variable in line 50 that can accept a string. We can avoid this by using a string variable N$ in place of N, and testing it and converting it to number form before we use it, but that's for when your programming has come on apace.

Now this type of loop allows you much more freedom than a FOR . . NEXT loop, because you are not confined to the use of a number to decide how many repetitions you have. The key to it is the use of IF to make a decision – and that's what we need to look at more closely now.

Decisions, decisions

We can make a number of types of comparisons between number variables or numbers, and these are listed in Figure 4.12. The mathematical signs are used for convenience, and you have to remember which way round the 'greater than' and 'less than' signs have to be. Remember, too, that signs like <>, <= and >= have to be put into the program using two separate keys for each. The keys of the Spectrum 128 that are marked with the combined signs can be used only when you are working in 48 BASIC mode. It's important also to note that the equals sign means 'identical to' when it is used in a test like this. If A is 3.9999999 and B is 4.0000000 then a test such as `IF a = b` will fail – A is not identical to B, even though it is close enough to be equal to our eyes. The important point here is that the numbers we see on the screen have been rounded, so that PRINT A in the example above might give the result 4. The test, however, is made on the numbers which have not been rounded.

Sign	Meaning
=	**Exact equality**
>	**left-hand quantity greater than right-hand quantity**
<	**left-hand quantity less then right-hand quantity**

The signs can be combined as follows:

<>	**Quantities not equal**
>=	**LHS greater than or equal to RHS**
<=	**LHS less than or equal to RHS**

***Note:* When the < or > sign is combined with =, then the < or > sign *must* be used first. Using a combination like => will always cause an error message when you try to enter the line.**

Figure 4.12 The mathematical signs that are used for comparing numbers and number variables.

```
 10 CLS
 20 PRINT "Press the y or n key
"
 30 PRINT " -Then press ENTER."
 40 LET repeat=40: LET exit=100
 50 INPUT a$
 60 IF a$="y" THEN PRINT "That'
s YES": GO TO exit
 70 IF a$="n" THEN PRINT "That'
s NO": GO TO exit
 80 PRINT "y or n only, please"
 90 GO TO repeat
100 REM
```

Figure 4.13 A simple y or n entry program, which tests string variables.

Figure 4.13 shows another test – this time on string variables. The instructions are in lines 20 to 30 , you are asked to type the `y` or `n` key. Line 40 starts a loop to get your answer, with an INPUT in line 50. You are expected to type `Y` or `N` and then press ENTER. The key that you have pressed has its value assigned to `a$`, so that `a$` should be `y` or `n`. Lines 60 to 80 then analyse this result. If the key that you pressed was `Y`, then line 60 prints 'That's YES', and the `GO TO exit` takes effect. The line that is used for `exit` carries a REM, because you can't have a line number by itself, and you can't have a GO TO with no line number. If `a$` is not `y`, however, line 60 is skipped, and the next test in line 70 is tried. If `a$` is `n`, the message in line 70 is printed, and the `GO TO exit` once more stops the program. If, however, you have pressed neither `y` nor `n`, both of these lines will be skipped, and the program moves to line 80. This patiently tells you that you have to use `y` or `n`, and since the next line is the `GO TO repeat` line, it sends the program back to the start of the loop at line line 40 again.

The test in this example is for identity. Only if `a$` is absolutely identical to `y` will the phrase `That's YES` be printed. If you typed a space ahead of `y`, or a space following it, or typed `Y` in place of `y`, then `a$` will `not` be identical, and the test fails. Failing means that `a$` is not identical to `y` and so everything that follows THEN in that line will be ignored. It's up to you to form these tests so that they behave in the way that you want!

Line 80 constitutes a 'mugtrap', a way of trapping mistakes. Very often when you have a choice of answers, you want to be sure that only certain replies are permitted. A mugtrap is a section of program that is intended to deal with an incorrect entry. A good mugtrap should show the user the error of his/her ways, and indicate what answer or answers might be more acceptable. This is very often important, because an incorrect entry in some types of program could cause the program to stop with an error message showing. For the skilled programmer, this is just a minor annoyance, but for the inexperienced user it can cause a minor panic. A good program doesn't allow any entries that would cause the program to stop, because when a program stops, you *could* lose all the information that you had typed into it. You don't have to – you can type GO TO, followed by the correct line number, then press ENTER. This can allow you to get back to normal – if you know what line number you need. An inexperienced user would not know this, and unless you had designed the program yourself and had a printed listing, even you might not know either. Mugtraps are our method of ensuring that mistakes do not stop a program.

```
  10 LET repeat=10: LET exit=110
  20 CLS : LET x=1+INT (RND*10)
  30 PRINT AT 2,7;"Guess the Num
ber!"
  40 PRINT AT 4,2;"If you get ne
ar, I'll tell you"
  50 INPUT n
  60 IF n=x THEN PRINT "Spot on!
": GO TO exit
  70 IF ABS (n-x)<3 THEN PRINT "
Close - it was ";x: GO TO exit
  80 PRINT "Nowhere near! Try ag
ain"
  90 FOR j=1 TO 300: NEXT j
 100 GO TO repeat
 110 PRINT "End of game"
```

Figure 4.14 A simple number-guessing game that tests both for equality and for a nearly-correct answer.

Just to emphasise the sort of power that these simple instructions give you, Figure 4.14 illustrates a very simple number-guessing game. Line 10 starts the loop which will be used when incorrect answers are given, and also assigns the line number for the end of a (successful) guess. Line 20 then clears the screen, and the `LET x =1+INT(RND*10)` step causes variable `x` to take a whole-number value that lies between 1 and 10. We can't predict what this value will be, because RNDmeans 'select at random'. RND picks numbers randomly enough for some games purposes, but not quite randomly enough for serious statistical users, and you should put a RANDOMIZE instruction before this line if you want to ensure that the numbers do not always follow the same sequence. In lines 30 and 40, the instructions ask you to guess the size of the number, with the difference that you don't have to find it exactly. You enter your number at line 50, and the tests are made in lines 60 and 70. If the number that you picked is identical to the random number, then you get the `Spot on` message in line 60, and the program ends because of the `GO TO exit` at the end of this line. The less obvious test is in line 70. The expression `n-x` is the difference between your guess, `n`, and the random number `x`. If your guess is larger than the number, then `n-x` is a positive number. If your guess is less than `x`, then `n-x` is a negative number. The effect of ABS, however, is to make any number positive, so that if `x` were 5 and you guessed 6 or 4, then `ABS(n-x)` would come to 1. If you get a difference of 1 or 2 (less than 3), the message in line 70 is printed, and the program ends once again. If you don't get anywhere near with your guess, the program repeats because of the GO TO in line 100. It's all very simple, but quite effective. You can probably see several flaws, however. Why should the program end after each successful attempt? Why not stay in the loop unless a zero is entered? How about devising a scoring system, using a number variable called 'score'? You could score 3 for a 'Spot on' and 1 for a 'Close', for example. Try working on this one to incorporate your own ideas, because its only by working on programs for yourself that you really learn what programming is all about.

Single key reply

So far, we have been putting in y or n replies with the use of INPUT, which means pressing the key and then pressing ENTER. This has the advantage of giving you time for second thoughts, because you can delete what you have typed and type a new letter before you press ENTER. For snappier replies, however, there is an alternative in the form of INKEY$. INKEY$ is an instruction that carries out a check of the keyboard to find if a key is pressed. This checking action is very fast, and normally the only way that we can make use of it is by placing the INKEY$ instruction in a loop of its own which will repeat until a key is pressed. Figure 4.15 shows such a loop. The INKEY$ instruction will produce a string quantity when any key is pressed, so we assign INKEY$ to a string variable, k$. In this way, when any key is pressed, the quantity that it represents will be assigned to k$, and if k$ is a 'blank string', meaning that no key was pressed, the line loops back to its start again. Note how we indicate a blank string by using two quotes with no space between them. By using the program of Figure 4.15 you can see the effect of pressing different keys. A few keys, such as the spacebar, will produce no visible character on the screen in line 40, but will nevertheless allow the program to jump out of its loop in line 30. A few keys, such as the EDIT, TRUE VIDEO and INVERSE VIDEO keys on the left-hand side of the keyboard, produce a question mark on the screen when they are pressed, and others, like the SHIFT keys, have no effect at all.

```
10 CLS
20 PRINT "Press any key"
30 LET k$=INKEY$: IF k$="" THE
N GO TO 30
40 PRINT "The key was ";k$
```

Figure 4.15 Using INKEY$ in a loop to find when a key has been pressed. Some keys will cause the program to operate, but will not print anything on the screen.

Chapter 5

Strings attached

String functions

In Chapter 3, we took a fairly brief look at number operations and functions. If numbers turn you on, that's fine, but string functions are in many ways more interesting. What makes them that way is that the really eye-catching and fascinating actions that the computer can carry out are so often done using string functions. What's a string function, then? As far as we are concerned, a string function is any action that we can carry out with strings. That definition doesn't exactly help you, I know, so let's go into more detail.

A string, as far as the Spectrum 128 is concerned, is a collection of characters which is represented by a string variable, a single-letter name that ends with the dollar sign. You can pack almost as many characters as you like into a Spectrum 128 string – many computers permit a maximum of only 255 characters per string, but the Spectrum 128 allows you up to 32767, and that's a longer string than most of us will ever need. Like other computers, too, the Spectrum stores its strings in a special way, making use of what is called ASCII code. The letters stand for American Standard Code for Information Interchange, and the ASCII (pronounced Askey) code is one that is used by most computers. Figure 5.1 shows a printout of the ASCII code numbers and the characters that they produce on my printer (EPSON RX–80).

Each character is represented by a number, and the range of numbers is from 32 (the space) to 127. On the printer, 127 produces a black square, but on the Spectrum screen, you'll see the copyright symbol (©). You can assign characters to a string variable by using the equality sign, with the use of LET. When you assign in this way, you need to use quotes around the characters. You can also assign using INPUT, when no quotes are needed. You can also assign using READ . . DATA and we have already looked at an example of this in Figure 3.12.

Now all number variables are represented in a different type of coding, one that uses the same number of codes no matter whether the value of the variable is large or small. Because a string consists of a set of number codes in the memory of the

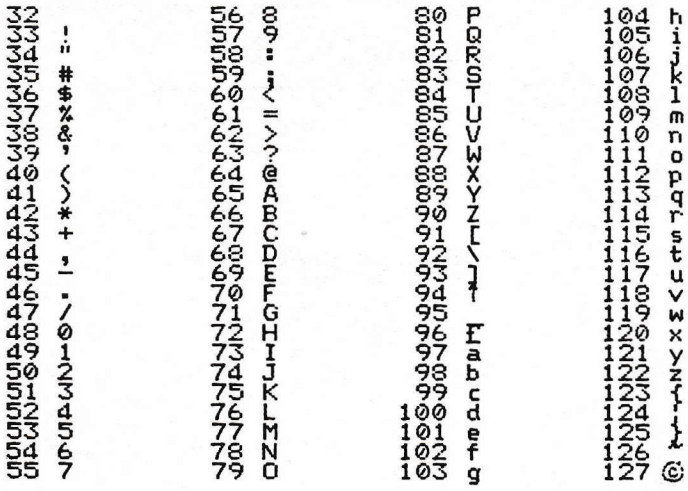


Figure 5.1 The standard ASCII code numbers.

computer, one code for each character, we can do things with strings that we cannot do with numbers. We can, for example, easily find how many characters are in a string. We can select some characters from a string, or we can change them or insert others. We can also convert from number coding into string coding, or the other way round. Actions such as these are the actions that we call *string functions*.

Len strikes again

One of these string function operations that I mentioned was finding out how many characters are contained in a string. Since a string can contain up to 32767 characters, an automatic method of counting them is rather useful, and LEN is that method. LEN has to be followed by the name of the string variable and the result of using LEN is always a number so that we can print it or assign it to a number variable. You can either put brackets around the name of the string, or separate it from LEN with a space. Using brackets makes the action more obvious, and it's the method that other computers use, so it's as well to get used to it, even though it means typing another character.

Figure 5.2 shows a simple example of LEN in use. Line 20 assigns a variable and line 30 tells you how many characters are in this variable. Note the word 'characters', it's not the same as 'letters'. Each space, full stop, comma and so on counts as a character for the purposes of a string, because each one is represented by its own ASCII code number. Lines 40 and 50 illustrate how you can find the length of a string which you have entered. This is something that is useful if you want to ensure that a name entered at the keyboard is not too long for the computer to use. You might, for example, have a program that places names in a form, with columns of restricted width, such as fifteen characters. If too long a name is entered, it could spill over into another column. You'll probably notice, if you have a long name or address, that when you get letters that have been computer-addressed, some part of the name or address may have been shortened. Now you know why, and how!

```
  10 CLS
  20 LET a$="Last of the Spectra
"
  30 PRINT "There are ";LEN (a$)
;" characters in the"'" phrase_
";a$
  40 INPUT "try one for yourself
"';k$
  50 PRINT
  60 PRINT k$;" has ";LEN k$;" c
haracters."
```

Figure 5.2 Introducing LEN, a member of the string function family.

All this is hardly earth-shattering, but we can turn it to very good use, as Figure 5.3 illustrates. This program uses LEN as part of a routine which will print a string called `t$` centered on a line. The example uses `PRINT AT`, but `PRINT TAB` is just as useful. This is an extremely useful routine to use in your own programs, because its use can save you a lot of tedious counting when you write your programs. The principle is to use LEN to find out how many characters are present in the string `t$`. This number is subtracted from 32, and the result is then divided by two. If the number of characters in the string is an odd number, then the number `c` will contain a .5, but this is completely ignored by AT when the string is printed. You can, incidentally, use 31 or 32. Whichever one you use, you will find that words are reasonably well centered – 32 works better with phrases which have an even number of letters, and 37 works better with phrases which have an odd number of letters. Yes, you could program that – but not yet!

The whole simple process could be done in one line, but I have shown it in three lines so that you can see what the steps are. We can use a routine of this type to centre anything that has the name `t$`. In Chapter 6, we'll be looking at the idea of 'Subroutines', which allow you to type the set of instructions (for centering a title, for example) just once, and then use them for any string that you like. This along with defined functions allow you a lot of control over what the machine can do.

There are two more string functions that are used for number conversions. Suppose, for example, that you have a number in the form of a string variable `v$`. This might have been obtained from a line such as:

```
INPUT v$
```

– and if you want to carry out arithmetic on the number it will have to be converted to number-variable form. This is done using VAL. If you have a line:

```
LET v=VAL(v$)
```

or

```
LET v=VAL v$
```

– then the number variable `v` will contain whatever number was in `v$`. The snag with this, as we'll see later, is that if the string `v$` contains anything that is not a number, the VAL action will not work, and the program will stop with an error message. The opposite action of conversion from number variable form to string variable form is much less troublesome. This action uses STR$. If, for example, you have a number in variable form `v`, then a line such as `LET v$=STR$ v` will convert the number into string form. You will see this action being used in music strings in Chapter 10.

```
10 CLS
20 LET t$="The Spectrum 128 in
action"
30 LET c=(32-LEN (t$))/2
40 PRINT AT 2,c;t$
```

Figure 5.3 Using LEN to print titles that will automatically be centered.

A slice of the action

The next group of string operations that we're going to look at are called slicing operations. The result of slicing a string is another string, a piece copied from the longer string. String slicing is a way of finding what letters or other characters are present at different places in a string. There's also a variation on this theme which allows you to insert new bits into a string.

All of that might not sound terribly interesting, so take a look at Figure 5.4. The string `a$` is assigned in line 20, and another string `b$` is assigned in line 30. The two strings are then printed in line 40, just to show there is no deception, ladies and gentlemen. There is nothing special here. In line 60, however, a new string is assigned, and when you see it printed in line 70, it produces `Spectrum`. Now how did this happen? The key to it is the use of items like `$(1)` and `b$(8)`. The `1` of `a$(1)` refers to the position of characters in the string, numbered from the start onwards. The character `a$(1)` is therefore the `S` of `Simple`, and `b$(8)` is the `m` of `many`. The letters are picked out in the same way, except where we need to take two letters together. Here we use `a$(11 TO 12)`, meaning that we take both the letters between places 11 and 12 in `a$`, so getting the `tr` part of the name.

```
10 CLS
20 LET a$="Simple controls".
30 LET b$="unlike many others"
40 PRINT a$;" ";b$
50 PRINT : PRINT
60 LET c$=a$(1)+a$(4)+a$(6)+a$
(8)+a$(11 TO 12)+b$(1)+b$(8)
70 PRINT c$
```

Figure 5.4 Extracting letters from strings by using a 'position number'.

Copying one letter of a string into another string is not terribly useful, though it's handy if you want to extract a set of initials from a name. The use of TO to copy more than one letter is a lot more useful. You need at most two numbers, one on each side of the word TO. Of these, one is the position number for the first letter of the sample, and the other number represents the position of the last letter. You don't however, need to use both numbers if you are slicing either from the left-hand side or fro the right. Figure 5.5 shows how you can slice a piece from the left-hand side of a string, using TO with no number preceding it.

```
10 CLS
20 LET a$="longitudinal"
30 PRINT a$( TO 4)
```

Figure 5.5 Using the string slicing action to get letters from the left-hand side of a string.

String slicing with TO and a single number isn't confined to copying a selected piece of the left-hand side of a string . We can also take a copy of characters from the right-hand side of a string. This particular facility isn't used quite so much as the left-hand copying one, but it's useful none the less. Figure 5.6 illustrates the use of the right-hand slice action to make use of the last word in a phrase. There are more serious uses than this. You can, for example, extract the last four figures from a string of numbers like 010–242–7015. I said a string of numbers deliberately, because something like this has to be stored as a string variable rather than as a number. If you try to assign this to a number variable, you'll get a silly answer. Why? Because when you type `LET n = Ø1-242-7Ø16` then the computer assumes that you want to subtract 242 from 10 and 7016 from that result.The value for `n` is –7248, which is not exactly what you had in mind! If you use `LET n$="Ø1Ø-242-7Ø16"` then all is well. When there is nothing following the TO in the brackets, the computer takes it that you want all the characters to the end of the string.

```
10 CLS
20 LET a$="That old Black Magi
c"
30 PRINT "Spectrum "+a$(16 TO
```

Figure 5.6 Using the slicing action to extract letters from the right-hand side of a string.

Now we can get quite a lot of interesting effects from these slicing methods, particularly when we start to use them in loops. Take a look at Figure 5.7 for an example, which does odd things with the letters of your name. The program prompts you to enter your name in line 20, and the name is assigned to `n$`. In line 30, we use LEN so that the number variable `n` contains the total number of characters in your name. This will include spaces and hyphens – nobody's likely to use asterisks and hash marks! Line 40 then starts a loop which uses the total number of characters as its end limit. Line 50 is the action line. When `j` is 1, line 50 prints the first letter on the left of your name on to the left-hand side of the screen, and the rest of your name a couple of spaces away. This depends, of course, on how many spaces you typed between the quotes in line 50. On the next pass through the loop, a new line is selected, and two letters are printed on the left side, with two fewer on the right side. This continues until the entire name is printed on the left, and only the last letter on the right.

```
20 INPUT "Your name, please";n
$
30 LET n=LEN n$
40 FOR j=1 TO n
50 PRINT n$( TO j);"  ";n$(j T
O )
60 NEXT j
```

Figure 5.7 Using slicing actions in a loop to print out words in a very odd way!

```
  10 CLS
  20 LET a$="Reference No. PDQ12
3 #7146"
  30 PRINT "Stores Ref. is ";a$(
15 TO 20)
```

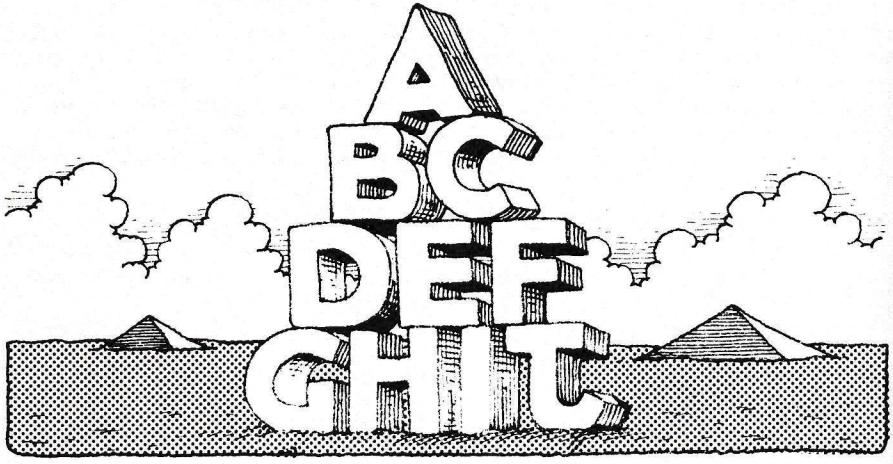
Figure 5.8 Slicing from any part of a string, using a number before and following TO. These numbers can, of course, be replaced by variables.

```
  10 LET a$="Sandstone"
  20 PRINT a$
  30 LET a$(8 TO 9)="rm"
  40 PRINT a$
```

Figure 5.9 Inserting new letters into a string. This will not be done if it might make the string longer.

```
  10 CLS
  20 INPUT "Your name, please ";
n$
  30 LET L=LEN n$: LET c=INT L/2
  40 FOR n=1 TO c
  50 PRINT AT n,16-n;n$(c-n+1 TO
 c+n)
  60 NEXT n
```

Figure 5.10 Making a letter pyramid to show the action of slicing with an expression.



Middling along

The main use of TO in string slicing is with two numbers, within brackets as usual, specifying the starting letter and the the ending letter of the piece that you want to slice. It's a lot easier to see in action that to describe, so try the program in Figure 5.8. Line 20 assigns a phrase to a$, and line 30 prints another phrase, making use of a set of reference letters and digits from the original phrase. It's a very simple example, but it shows TO being used to extract from the middle of a string, which is what it's all about.

This slicing business, however, can be a two-way trade, as Figure 5.9 illustrates. Line 10 allocates a string variable to the word SANDSTONE. This is printed, and then line 30 inserts two letters into the string. This makes no difference to the length of the string, it simply means that letters 8 and 9 are RM in place of NE'. When the string is printed again in line 40, it shows the change. What would happen if you tried to insert too much? Try it for yourself – make line 30 read:

```
30 LET a$(8 TO 10)="rms"
```

and run this. As you'll see, you get the original word sandstone now, and an error message:

```
Subscript wrong, 30:1
```

– meaning that the string would be too long if this were done. The important point about a Spectrum string is that you cannot extend it except by tacking another piece on with the + operation. Can you make a string shorter in this way? Type another line 30:

```
30 LET a$(5 TO 9)=""
```

and run this. The word that is printed in line 40 is now sand, so the word has been shortened – but the string has not! If you type PRINT LEN(a$), you'll get the result of 9, so that what has happened is that each letter following the d of sand has been replaced by a space. This can cause a lot of problems if you are using the length of strings for anything important, like centering titles or placing strings into columns.

One of the features of all of these string slicing instructions is that we can use variable names or expressions in place of numbers. Figure 5.10 shows a more elaborate piece of slicing which uses expressions. It all starts innocently enough in line 20 with a request for your name. Whatever you type is assigned to variable n$, and in line 30 a bit of mathematical juggling is carried out. How does it work? Suppose you type as your name DONALD. This has six letters, so in line 30 variable L is assigned to 6, and c is the whole number part of L/2 (equal to 3). Line 40 then starts a loop of 3 passes. In the first pass you print at position 1,15 – because n=1 and 16-n=15. What you print is the name n$(3 TO 4). With n=1 and c=3, then c-n+1 is 3-1+1, which is 3, and c+n is 3+1, which is 4. What you print, then, is the third and fourth letters of the name. On the next run through the loop, n is 2, c-n+1 is 2, and c+n is 5. What is printed is n$(3 TO 5), which is ONAL. The loop goes on in this way, and the result is that you see on the screen a pyramid of letters formed from your name. It's quite impressive if you have a long name! If your name is short, try making up a longer one.

More priceless characters

It's time now to look at some other types of string functions. If you hark back a few pages, you'll remember that we introduced the idea of ASCII code. This is the number code that is used to represent each of the characters that we can print on the screen. We can find out the code for any letter by using the function CODE, which is followed, either within brackets or after a space, by a string character in quotes or a string variable (with no quotes). The result of CODE is a number, the ASCII code number for that character. If you use `PRINT CODE("Spectrum 128")`, then you'll get the code for the `S` only, because the action of CODE includes rejecting more than one character. Figure 5.11 shows this in action. String variable `a$` is assigned in line 10 and in line 30 a loop starts which will run through all the letters in `a$`. The letters are picked out one by one, using `a$(n)`, and the ASCII code for each letter is found with CODE. The space between quotes, along with the semicolons in line 40 makes sure that the codes are all printed with a space between the numbers, and without taking a new line for each number. Simple, really, and very useful if you want to find codes without having to look them up in the manual.

CODE has an opposite function, CHR$. What follows CHR$, within brackets or after a space, has to be a code number, and the result is the character whose code number is given. The instruction `PRINT CHR$(65)` (or `PRINT CHR$ 65`), for example, will cause the letter `A` to appear on the screen, because 65 is the ASCII code for the letter A. We can use this for coding messages. Every now and again, it's useful to be able to hide a message in a program so that it's not obvious to anyone who reads the listing. Using ASCII codes is not a particularly good way of hiding a message from a skilled programmer, but for non-skilled users it's good enough. Figure 5.12 illustrates this use. Line 50 contains an INKEY$ loop to make the program wait for you. When you press a key, the loop that starts in line 60 prints 19 characters on the screen. Each of these is read as an ASCII code from a list, using a READ . . DATA instruction in the loop. The `PRINT CHR$ n` in line 70 then converts the ASCII codes into characters and prints the characters, using a semicolon to keep the printing in a line. Try it!. If you wanted to conceal the letters more thoroughly, you could use quantities like one quarter of each code number, or 5 times each code less 20, or anything else you like. These changed codes could be stored in the list, and the conversion back to ASCII codes made in the program. This will deter all but really persistent decoders! This example, incidentally, illustrates the use of READ and DATA in a loop. We would normally use READ and DATA only for information that we particularly wanted to keep stored in a program like this. Another point is the use of RESTORE in line 30. This does not need to be included in such a simple example, but as we noted earlier in Chapter 3, it is needed if the codes have to be read again in the same program.

```
 10 LET a$="Spectrum 128 comput
ing"
 20 CLS : PRINT
 30 FOR n=1 TO LEN a$
 40 PRINT CODE a$(n);" ";
 50 NEXT n
```

Figure 5.11 Using CODE to find the ASCII code for letters.

```
  10 CLS
  20 PRINT AT 2,2;"What's the me
aning of spectrum?"
  30 RESTORE
  40 PRINT AT 4,2;"Press any key
 for the answer."
  50 LET k$=INKEY$: IF k$="" THE
N GO TO 50
  60 FOR j=1 TO 19
  70 READ n: PRINT CHR$ n;
  80 NEXT j
  90 DATA 99,111,108,111,117,114
,115,32,105,110,32,115,101,113,1
17,101,110,99,101
```

Figure 5.12 Using ASCII codes to carry a coded message, and then using CHR$ to obtain the character that corresponds to a code number.

While we are on the subject of RESTORE, there's another twist to this instruction in the form of placing a line number following RESTORE. RESTORE with a line number means that the DATA list will start again from the beginning of that line. You must make sure, of course, that the line number which you have chosen is a line that starts with DATA! Take a look at Figure 5.13. This offers a choice of data to be read by making use of RESTORE along with a number. When you pick a number, it is used in line 60 to carry out a RESTORE command which has a line number equal to one thousand times your selected number following it. When a is 2, for example, RESTORE 1000*a means 'start reading DATA at line 2000'. Each DATA line contains four string items, so that when lines 70 to 80 are carried out, four items will be read from whichever line has been picked. It's a useful way of selecting from a number of lists which will be used each time the program is used.

```
  10 CLS
  20 PRINT "Which list do you wa
nt?"
  30 LET repeat=30
  40 INPUT "Number 1 to 3 please
 ";a
  50 IF a<1 OR a>3 THEN PRINT "I
ncorrect selection": GO TO repe
at
  60 RESTORE a*1000: PRINT
  70 FOR n=1 TO 4
  80 READ x$: PRINT x$;" ";: NEX
T n
  90 STOP
1000 DATA "Metro","Maestro","Mon
tego","Rover"
2000 DATA "Nova","Astra","Belmon
t","Cavalier"
3000 DATA "Fiesta","Escort","Ori
on","Sierra"
```

Figure 5.13 How RESTORE can be used to select different DATA lines.

The law about order

We saw earlier, in Figure 4.12, how numbers can be compared using signs such as =, < and >. We can also compare strings with these signs, using the ASCII codes as the basis for comparison. Two letters are identical if they have identical ASCII codes, so it's not difficult to see what the identity sign (=) means when we apply it to strings. If two long strings are identical, then they must contain the same letters in the same order. It's not so easy to see how we use the > and < signs until we think of ASCII codes. The ASCII code for A is 65, and the code for B is 65. In this sense, A is 'less than' B, because it has a smaller ASCII code. If we want to place letters into alphabetical order, then, we simply arrange them in order of ascending ASCII codes.

This process can be taken one stage further, though, to comparing complete words, character by character. Figure 5.14 illustrates this use of comparison using the = and > symbols. Line 20 assigns a nonsense word – it's just the first six letters on the top row of letter keys. Line 30 then asks you to type a word. The comparisons are then carried out in lines 40 and 50. If the word that you have typed, which is assigned to `b$` is identical to `qwerty`, then the message in line 40 is printed, and the program ends. If `qwerty` would come later in an sorted list than your word, then line 50 is carried out. If, for example, you typed `peripheral`, then since `q` comes after `p` in the alphabet and has an ASCII code that is greater than the code for `p`, your word `b$` scores lower than `a$`, and line 50 swaps them round. This is done by assigning a new string, `q$` to `a$` (so that `q$` is `qwerty`), then assigning `a$` to `b$` (so the `a$` is `peripheral`), then `b$` to `q$` (so that `b$` is `qwerty`). Line 60 will then print the words in the order A$ and then B$, which will be the correct alphabetical order. If the word that you typed comes later than `qwerty`, such as `tape`, then `a$` is not 'greater than' `b$`, and the test in line 50 fails. No swap is made, and the order `A$`, then `B$`, is still correct. Note the important point though, that words like `qwertz` and `qwertx` will be put correctly into order – it's not just the first letter that counts. What you *do* have to be careful about is words that mix upper- and lower-case letters, because in the ASCII code list, all of the capitals have lower code numbers than the lower-case letters. This would mean that a word such as `ZERO` would be put ahead of the word `advance` if you used ASCII codes by themselves to put the words into order.

```
  10 CLS
  20 LET a$="qwerty"
  30 PRINT : INPUT "Type a word
";b$
  40 IF b$=a$ THEN PRINT "Same a
s mine!": STOP
  50 IF a$>b$ THEN LET q$=a$: LE
T a$=b$: LET b$=q$
  60 PRINT "Correct order is ";a
$;" then ";b$
  70 STOP
```

Figure 5.14 Comparing words to decide on their alphabetical order.

Lists

The variable names that we have used so far are useful, but there's a limit to their usefulness. Suppose, for example, that you had a program that allowed you to type in a large set of numbers. How would you go about assigning a different variable name to each item? The answer is that this would be an impossible task, it would make the program hopelessly clumsy; and we need a way around the problem. Figure 5.15 illustrates this. Lines 10 to 40 generate an (imaginary) set of examination marks. This is done simply to avoid the hard work of entering the real thing! The variable `a(n)` in line 30 is something new, though. It's called a *subscripted variable*, and the *subscript* is the number that is represented by `n`. The name that we use has nothing to do with computing, it's a name that was used long before computers were around. How often do you make a list with the items numbered 1, 2, 3 . . ., and so on? These numbers 1, 2, 3 are a form of subscript number, put there simply so that you can identify different items. Similarly, by using variable names `a(1)`, `a(2)`, `a(3)` and so on, we can identify different items that have the common variable name of `a`. A member of this group like `a(2)` has its name pronounced as 'a-of-two'.

The usefulness of this method is that it allows us to use one single variable name for the complete list – an *array*, to give it its other name – picking out items simply by their identity numbers. Since the number in the brackets can be a number-variable or an expression, this allows us to work with any item of the list. Figure 5.15 shows the list being constructed from the FOR . . NEXT loop in lines 20 to 40. Each item is obtained by finding a random number between 11 and 100, and is then assigned to `a(n)`. Ten of these 'marks' are assigned in this way, and then lines 60 to 90 print the list. It makes for much neater programming than you would have to use if you needed a separate variable name for each number.

So far, so good, but one point has been omitted so far. The first line contains the instruction `DIM a(10)`. If you leave this out, you will find that the program can't run, and you get the error message `2 Variable not found, 30:1` whenever the computer meets the variable name of `a(n)`. The computer has to be prepared for the use of this type of variable name – the preparation consists of getting some memory ready to receive the data. When you use `DIM` (meaning 'dimension'), the memory is allocated for the array. A line such as `DIM a(10)` allows you up to ten items in the list, numbered `a(1)` to `a(10)` – a lot of other computers allow you to use `a(0)` as well, so that you have 11 items dimensioned by `DIM a(10)`. You must not attempt to use `A(11)` or any higher number, if you have used `DIM a(10)` otherwise you will get the error message `3 Subscript wrong 30:1`.

```
10 CLS : DIM a(10)
20 FOR n=1 TO 10
30 LET a(n)=11+INT (RND*90)
40 NEXT n
50 PRINT
60 PRINT AT 2,11;"Marks List"
70 PRINT : FOR n=1 TO 10
80 PRINT AT n+3,2;"Item ";n;"
received ";a(n);" marks."
90 NEXT n
```

Figure 5.15 An array of subscripted number variables. It's simpler than the name suggests!

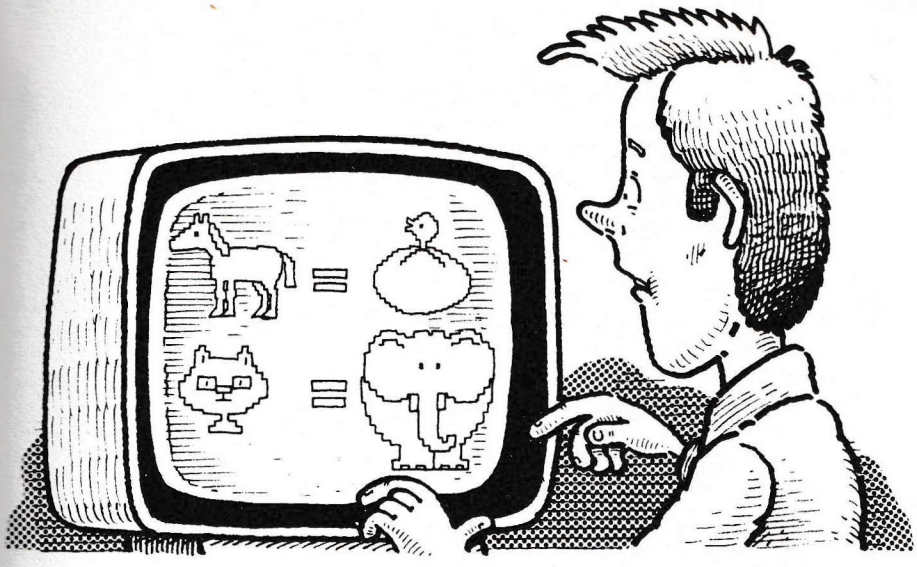
```
  10 CLS : DIM a(10): DIM n$(10,
20)
  20 PRINT AT 2,1;"Please enter
names and marks"
  30 FOR n=1 TO 10
  40 INPUT "Name ";n$(n)
  50 INPUT "Mark ";a(n)
  60 NEXT n
  70 CLS : LET total=0
  80 PRINT AT 2,11;"MARKS LIST"
  90 FOR n=1 TO 10
 100 PRINT AT n+3,2;n$(n)
 110 PRINT AT n+3,16;a(n)
 120 LET total=total+a(n)
 130 NEXT n
 140 PRINT
 150 PRINT "Average is ";total/1
0
```

Figure 5.16 Using strings in one array, and numbers in another. The arrays have been 'dimensioned', and the string array needs two numbers within its DIM brackets.

The important DIM instruction, then, consists of naming each variable that you will use for arrays, and following the name with the maximum number, within brackets, that you expect to use. You aren't forced to use this number, but you certainly must not exceed it. If you do, and your program stops with an error message, you will have to change the DIM instruction and start again – which will be tough luck if you were typing in a list of 100 names, because changing the DIM number will automatically erase whatever array was present before!

Figure 5.16 extends this use of array variables another step further. This time you are invited to type a name and a mark for each of ten items. When the list is complete, the screen is cleared and a variable called 'total' is set to zero in line 70. The list is then printed neatly, and on each pass through the loop the total is counted up (in line 120) so that the average value can be printed at the end. The important point here is that it's not just numbers that we can keep in this list form, and Figure 5.16 uses both a string array (of names) and a number array (of marks).

There is, however, an important difference between a string array and a number array when you use the Spectrum. A lot of other computers allow you to dimension a string array just as you dimension a number array. The Spectrum is different. For a string array like this, you need two numbers following the DIM. The first of these is, as before, the maximum item number that you will use, ten in this example. The second number is the maximum length of the string. In the example, this has been set to 20. If you enter any name that consists of more than 20 characters, then it will be chopped to a size of twenty characters. As it is, a name as long as 15 characters would cause the name to overlap the marks column, so this restriction can be turned to good advantage. It does mean, though, that you have to plan your use of string arrays much more carefully on the Spectrum than on most other machines.



Rows and columns

You can imagine an array as a list of items, one after the other, but there is a variety of array which allows a different kind of list, called a matrix. A matrix is a *list of groups* or items, with all the items in a group related. We could think of a matrix as a set of rows and columns, with each group taking up a row, and the items of a group in separate columns. Take a look at Figure 5.17 to see how this works. We use here a variable n$ which has three subscript numbers. The first number is the row number, the second is the column number, and the third is, as before, the maximum length of the string. We need two FOR . . NEXT loops to read data into this matrix. This is carried out in lines 20 to 50. The items are then printed in columns by the loop in lines 60 to 90. In this loop, the variable n is used as the row number and we use the column numbers 1 and 2. The rows contain animal names, and the columns separate the different names that we use for adult and for young animals respectively. This example has used a string matrix, but a number matrix is also possible, and it needs no length dimension number.

```
 10 CLS : DIM n$(3,2,10)
 20 FOR n=1 TO 3
 30 FOR j=1 TO 2
 40 READ n$(n,j)
 50 NEXT j: NEXT n
 60 FOR n=1 TO 3
 70 PRINT AT n+3,5;n$(n,1);
 80 PRINT AT n+3,16;n$(n,2)
 90 NEXT n
100 DATA "Horse","Foal","Cow","
Calf","Dog","Puppy"
```

Figure 5.17 Making a matrix of rows and columns.

```
  10 CLS : DIM a$(50,2,20): DIM
j$(1)
  20 FOR n=1 TO 50
  30 PRINT AT 8,2;"";: INPUT "Na
me ";a$(n,1)
  40 PRINT AT 10,2;"";: INPUT "T
el. No. ";a$(n,2)
  50 CLS : NEXT n
  60 CLS : PRINT AT 9,12;"LIST C
OMPLETE"
  70 PRINT AT 10,2;"";: INPUT "P
ick an initial letter ";j$
  80 FOR n=1 TO 50
  90 IF j$=a$(n,1)(1) THEN PRINT
 "Name - ";a$(n,1): PRINT "Numbe
r - ";a$(n,2)
 100 NEXT n
```

Figure 5.18 Using a name and number matrix for a simple telephone directory application.

```
10 CLS
20 DEF FN x$(a$,a)=a$( TO a)
30 LET a$="definition"
40 FOR n=1 TO LEN a$
50 PRINT FN x$(a$,n)
60 NEXT n
```

Figure 5.19 Illustrating a string defined function in action.

```
10 DEF FN c(a$)=16-LEN a$/2
20 LET x$="TITLE"
30 PRINT TAB FN c(x$);x$
```

Figure 5.20 A string defined function that can be used for centering a title. The advantage of using a defined function is that you can use any name for the title.

Figure 5.18 shows a much more ambitious matrix program. The idea is to store sets of names and telephone numbers which are fed in by you in the course of the loop in lines 20 to 50. Once the matrix has been filled, you can pick an initial letter for a name, and ask the computer to print out the name and number that it has located. I've left out mugtraps just to keep this example reasonably short, but you would certainly need some sort of mugtraps, even if only in the form of a message like:

```
PRINT" Sorry, can't find ";j$;" entries"
```

What's new, then? Line 90 is the one to watch. You are trying to identify the first letter of a name, and the name is in the form of an array variable. The variable `j$`, remember, is a first letter that you have picked, and you want to find if `a$(1,1)` or `a$(2,1)` or `a$(3,1)`, etc., might start with this letter. To find the first letter of a string array item, you need the rather curious-looking statement `a$(n,1)(1)`. If you think about it, though, this is quite logical. To pick the first letter of a variable called `a$`, you would use `a$(1)`, and so to pick the first letter of a variable called `a$(n,1)`, you use `a$(n,1)(1)`. You could just as easily pick two letters by using `a$(n,1)(1 TO 2)`. In the dimensioning line, we have used DIM `j$(1)`. This has the effect of forcing `j$` to be of one character only, so that if you enter several characters, only the first is used. This is quite a neat dodge, because it avoids having to use `j$(1)` when the comparison is made. When you try this program out, use loops of 1 TO 5 rather than 1 TO 50 to save your typing finger(s). You'll find also that the program takes quite a long time to save or load. This is because of the array size of 50 items, because the Spectrum saves all the space that is reserved for the array, whether you have filled the array or not. We'll look at that point again later.

Loose end

There is one string dodge that we haven't looked at so far, the use of defined string functions. We saw earlier in Chapter 3 that we could use lines that started with DEF FN to define what we wanted to do with some numbers. We can also use DEF FN with string variables, and this gives us access to some quite useful tricks. When we use a string defined function, it needs to have a string name, and the action that is carried out following the equality sign can be any single string action. You can't do more than this, like having a loop, but you can take the CODE for the first letter, slice the string, or whatever. The action is illustrated in Figure 5.19. The defined function prints the left-hand side of a string `a$`, using a number of characters that is decided by the number assigned to variable `a`. To see it used, the program then assigns a word to `a$`, and then enters a loop that will make the number of characters one more in each pass through the loop. The advantage of using a defined function for actions of this type is that you need type the slicing instruction only once, and you can then use it with any string, just by putting the string name and the slice number into the `FN x$` part. Figure 5.20 illustrates another string defined function. This is a title centering function that can be used to centre any string of less than 32 letters.

Chapter 6

Menus, choices and design

Figure 4.13 introduced the idea of making a choice by pressing a key. In that example, the choice of keys was limited, `y` or `n`. A choice of two items isn't exactly generous, and we can extend the choice by a kind of program routine that is called a menu. A menu is a list of choices, usually of program actions. By picking one of these choices, we can cause one section of the program to be run. One way of making the choice is by numbering the menu items, and typing the number of the one that you want to use. Figure 6.1 shows what a typical menu of this type would look like on the

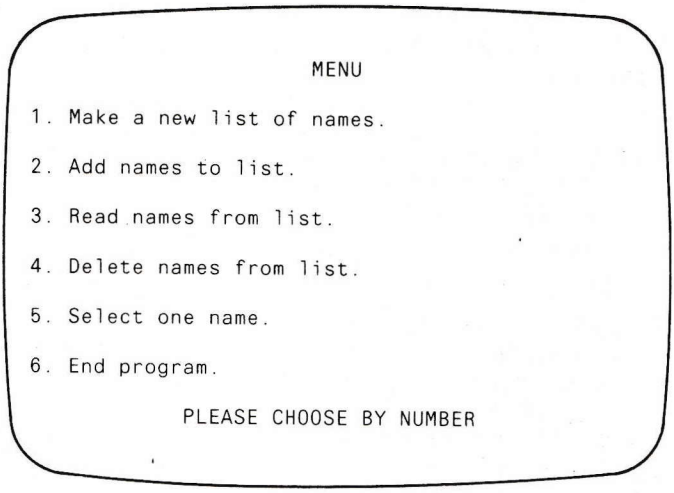


Figure 6.1 A typical menu as it would appear on the screen.

screen. The method of making the choice could be a very simple one, using a set of lines such as:

```
IF k=1 THEN GO TO 1000
IF k=2 THEN GO TO 2000
```

and so on, but as we shall see, the Spectrum 128 offers you a better method.

To start with, suppose we want to pick from four items by typing numbers 1 to 4 on the keyboard. The first thing that you have to ensure is that only the numbers 1 to 4 are accepted, not numbers like 0, 5, –10 and so on – in other words, a bit of mugtrapping. The second point is that the use of INPUT is a bit tedious, because you have to press ENTER after you have typed your number-key. We'll use INKEY$, then, to get the reply (the number-key), and it only remains to check that you have chosen a number that is in the correct range. At the same time, we want to make sure that the program doesn't stop with an error message if you enter a letter instead of a number, so we can add this action to the mugtrapping also. The easiest method of separating the desired numbers from anything else is to use CODE. The ASCII codes for numbers 1 to 4 are 49 to 53 inclusive, so that if we subtract 48 from the CODE for k$, we will get the number. If you pressed the wrong number key, then the result will not be in the range of 1 to 4. More important, if you pressed a letter key, the CODE number will be 65 or more, and this will also be rejected by the test. This is important, because the Spectrum 128 will not accept the line LET k=VAL k$ if k$ is not a number. By using CODE here, then, we avoid a possible problem when a letter key is hit by accident. The technique is illustrated in Figure 6.2.

```
 10 CLS : PRINT AT 2,14;"MENU"
 20 PRINT AT 2,2;"Please select
by number 1 - 4"
 30 LET choice=30
 40 LET k$=INKEY$: IF k$="" THE
N GO TO 40
 50 LET k=CODE k$-48: IF k<1 OR
 k>4 THEN PRINT "Incorrect choic
e of ";k$: FOR j
=1 TO 200: NEXT j: GO TO choice
 60 GO TO 100*k
 70 STOP
100 PRINT "Choice 1"
110 STOP
200 PRINT "Choice 2"
210 STOP
300 PRINT "Choice 3"
310 STOP
400 PRINT "Choice 4"
410 STOP
```

Figure 6.2 A menu choice which uses INKEY$ and CODE to obtain the selected number.

Now that the variable k holds a number in the correct range of 1 to 4, we can make the selection. This is done with incredible ease in line 60, using GO TO 1ØØ*k. As we noted earlier, this mean that if you choose 1, the program goes to line 100, if you choose 2 the program goes to line 200 and so on. The only thing that you have to watch is that you now place the correct bits of program into these lines. In the program, the first two lines print the title ENU, and show the choice of numbers. The choice action then starts in line 30, which opens a loop that will be used for getting the number choice. The first action in the loop is line 40, which is just an ordinary INKEY$ loop. If you press a key while this loop is operating, then the character code for that key is assigned to k$ and this is processed and tested in line 50. By using CODE k$-48 we extract a number, and then we test it for the correct range of 1 to 4. If the number is not in the correct range we print a message, wait for a short time, and then loop back. Line 60 is the selecting line, and the lines that it goes to contain simple PRINT instructions, and are followed by STOP lines. This has to be done, because if the STOP lines were omitted, then making choice 1 would give the messages for 1, 2, 3, and 4! You should use STOP wherever you want to make sure that the program cannot go on from one line to the next unintentionally.

Sectional programming

A lot of programs consist of a title, some instructions, and then a menu. Depending on what menu choice you have made, some part of the program is run, and the program ends, or returns to the menu again. The use of something like GO TO 1ØØ*k as we have now seen, can make it reasonably easy for you to write programs of this style, about which there's more in this chapter. The idea of having a program that consists of a lot of separate pieces is very important. Every version of BASIC contains commands that allow you to have pieces of programs, called subroutines, which can be called for and used as needed. The Spectrum 128 is no exception, and for the rest of this chapter, then, we'll be looking at how to make use of Spectrum *subroutines*.

To start with, a subroutine is a miniature piece of program that carries out part of the action of the complete program. Usually, this will be some action that is required over and over again, like getting a name or number from the user at the keyboard, or printing a warning message. The important point is that a subroutine can be started or *called* by a command in the main program, and after the subroutine has finished its work, the main program then continues on its way.

Figure 6.3 shows a very simple subroutine in use. The subroutine has been placed starting in line 100, so that the first line of the program defines the word centre as

```
 10 LET centre=100
 20 LET t$="This is a title"
 30 GO SUB centre
 40 STOP
100 LET st=16-LEN t$/2
110 PRINT TAB st;t$
120 RETURN
```

Figure 6.3 Using a subroutine – this is the key to more advanced programming.

meaning line 100. The subroutine will take a string called t$ and print it centered, so that lines 20 and 30 simply assign a phrase to t$ and then call the subroutine. This is done by using GO SUB centre, and we could just as easily have used GO SUB 1ØØ. Lines 100 and 110 define what the subroutine must do. This is completely straightforward, and the only thing that marks this piece of program out as being a subroutine is that it ends with the word RETURN, meaning return to the main program, or to whatever follows the GO SUB instruction that started the subroutine operating. One important point, however, is the use of STOP in line 40, because this is needed to prevent the lines of the subroutine from being carried out again after line 30 has run. Unintentional operation of a subroutine is called 'crashing through', and STOP prevents this.

Now, having looked at it, what's the point? One very useful point is that it greatly simplifies the design of programs, something that we shall be looking at very shortly. A lot of computer owners never get round to programming because they never know where to start. Even if they know the instruction words of BASIC, they can never quite see how they can use this knowledge to construct a program. That a tragedy, because as I've said, programming is what computers are for. If you buy a machine just to run other peoples programs, why spend so much? It's like buying an airliner because you want to fly to Torremolinos. Bought programs can be wonderful. They can also be costly, difficult to use, and require far too much effort to do something quite simple. If you can program for yourself, you can make the programs that *you* want, and that's what computing is really about.

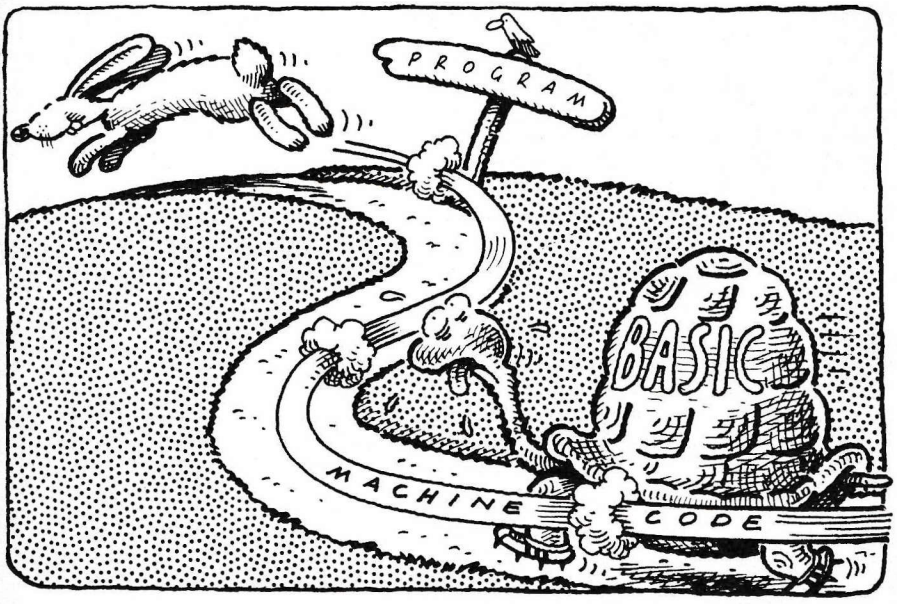
Returning to subroutines, then, we start a subroutine going by typing a GO SUB, followed by a line number or, as in the example, the variable name for a line number. Using variable names for line numbers makes it a lot easier to follow what a program is doing, and is a very great help in designing your own. It has one snag about it though. If your program is ever renumbered, using the very convenient RE-NUMBER choice on the EDIT menu of the Spectrum, then all the GO SUBs will be to the wrong numbers! This doesn't happen if you use line numbers directly, like GO SUB 1ØØØ, because the RENUMBER command automatically renumbers these too. When you have used LET centre=1ØØØ and GO SUB centre, though, the renumbering does not change the assignment of 1000 to centre, so all of the GO SUBs in a program will be made incorrect. You can get round this in two ways. One is by using only the GO SUB 1ØØØ form of the instruction, the other is by never renumbering except when it's absolutely essential. Take your pick.

Rolling your own

You can get a lot of enjoyment from your Spectrum computer when you use it to enter programs from cassettes that you have bought. You can obtain even more enjoyment from typing in programs that you have seen printed in magazines. Even more rewarding is modifying one of these programs so that it behaves in a rather different way, making it do what suits you. The pinnacle of satisfaction, as far as computing is concerned, however, is achieved when you design your own programs. These don't have to be masterpieces. Just to have decided what you want, written it as a program, entered it and made it work is enough. It's 100% your own work, and you'll enjoy it all the more for that.Now I can't tell in advance what your interests in programs might be. Some readers might want to design programs that will keep tabs on a stamp collection, a record collection, a set of notes on food preparation or the

technical details of vintage steam locomotives. Programs of this type are called *database* programs, because they need a lot of data items to be typed in and recorded. On the other hand, you might be interested in games, colour patterns, drawings, sound, or other programs that require shapes to move across the screen. Programs of that type need instructions that we shall look at in quite a lot of detail in chapters 8 to 10. What we are going to look at in this section is the database type of program, because it's designed in a way that can be used for all types of programs. Once you can design simple programs of this type, you can progress, using the same methods, to design your own graphics and sound programs. Remember, though, that most of the very fast moving or elaborate graphics programs that you see are not written in BASIC. The reason is that BASIC is too slow to allow fast movement, or the control of lots of moving objects. These arcade-type programs or the adventure types, like the two sample programs that come with your Spectrum, are written in 'machine-code', a set of number-coded instructions direct to the Z-80 microprocessor that is the heart of the computer. This bypasses BASIC altogether, and is very much more difficult. If you learn how to design programs in BASIC, however, you will be able to learn machine code later. All you need for BASIC is experience – a lot of it. For machine code you also need a considerable understanding of how the computer works, and a lot of information about the way its memory is used. That takes time and needs the study of a lot of books.

Two points are important here. One is that experience counts in this design business. If you make your first efforts at design as simple as possible, you'll learn much more from them. That's because you're more likely to succeed with a simple program first time round. You'll learn more from designing a simple program that works than from an elaborate program that never seems to do what it should. The second point is that program design has to start with the computer switched off, preferably in another room! The reason is that program design needs planning, and you can't plan properly when you have temptation in the shape of a keyboard in front of you. Get away from it!



Aims

1. **Present the name of an animal on the screen, picked at random.**
2. **Ask what its young is called.**
3. **Reply must be correctly spelled.**
4. **User must not be able to read correct answer from the listing.**
5. **Give one point for each correct answer.**
6. **Allow two chances at each question.**
7. **Keep a track of the number of attempts.**
8. **Present a score as number of correct answer out of number of attempts.**

Figure 6.4 A program outline plan. This is your starter!

Put it on paper

We start, then, with a pad of paper. For myself, I use an A4 'student's pad' which is punched so that I can put sheets into a file. This way, I can keep the sheets tidy, and add to them as I need. I can also throw away any sheets I don't need, which is just as important. Even a very simple program is probably going to need more than one sheet of paper for its design. If you then go in for more elaborate programs, you may easily find yourself with a couple of dozen sheets of planning and of listing before you get to the keyboard. Just to make the exercise more interesting, I'll take an example of a program, and design it as we go. This will be a very simple program, but it will illustrate all the skills that you need.

Start, then, by writing down what you expect the program to do. You might think that you don't need to do this, because you know what you want, but you'd be surprised. There's an old saying about not being able to see the wood for the trees, and it applies very forcefully to designing programs. If you don't write down what you expect a program to do, it's odds on that the program will never do it! The reason is that you get so involved in details when you starting writing the lines of BASIC that it's astonishingly easy to forget what it's all for. If you write it down, you'll have a goal to aim for, and that's as important in program design as it is in life. Don't just dash down a few words. Take some time about it, and consider what you want the program to be able to do. If you don't know, you can't program it! What is even more important is that this action of writing down what you expect a program to do gives you a chance to design a properly structured program. Structured in this sense means that the program is put together in a way that is a logical sequence, so that it is easy to add to, change, or re-design. If you learn to program in this way, your programs will be easy to understand, take less time to get working, and will be easy to extend so that they do more than you intended at first. The commands of the Spectrum make it easy to structure your programs properly.

As an example, take a look at Figure 6.4. This shows a program outline plan for a simple game. The aim of the game is to become familiar with the names of animals and their young. The program plan shows what I expect of this game. It must

1. **Display title, then instructions.**
2. **Display name of animal.**
3. **Ask for name of young.**
4. **Use INPUT for reply.**
5. **Compare reply with correct answer.**
6. **If correct, add 1 to score, and ask if another one is wanted.**
7. **If incorrect first reply, allow another try without incrementing number of tries.**
8. **If second reply incorrect, select another question.**
9. **Game ends when user types** `n` **in reply to** `Do you want another one.`

Figure 6.5 The next stage in expanding the outline.

present the name of an animal, picked at random, on the screen, and then ask what the name of its young is. A little bit more thought produces some additional points. The name of the young animal will have to be correctly spelled. A little bit of trickery will be needed to prevent the user (son, daughter, brother, or sister) from finding the answers by typing LIST and looking for the DATA lines. Every game must have some sort of scoring system, so we allow one point for each correct answer. Since spelling is important, perhaps we should allow more than one try at each question. Finally, we should keep track of the number of attempts and the number of correct answers, and present this as the score at the end of each game. Now this is about as much detail as we need, unless we want to make the game more elaborate. For a first effort, this is quite enough. How do we start the design from this point on?

The answer is to design the program in the way that an artist paints a picture or an architect designs a house. That means designing the outlines first, and the details later. The outlines of this program are the steps that make up the sequence of actions. We shall, for example, want to have a title displayed. Give the user time to read this, and then show instructions. There's little doubt that we shall want to do things like assign variable names, dimension arrays, and other such preparation. We then need to play the game. The next thing is to find the score, and then ask the user if another game is wanted. Yes, you have to put it all down on paper! Figure 6.5 shows what this might look like at this stage.

Foundation stones

Now at last, we can start writing a chunk of program. This will just be a foundation, though. What you must avoid at all costs is filling pages with BASIC lines at this stage. As any builder will tell you, the foundation counts for a lot. Get it right, and you have decided how good the rest of the structure will be. The main thing you have to avoid now is building a wall before the foundation is complete!

Figure 6.6 shows what you should aim for at this stage. There are only thirteen lines of program here, and that's as much as you want. This is a foundation, remember, not the Empire State Building. It's also a program that is being developed, so we

```
10 CLS : GO SUB 9000
20 REM get gosub numbers
30 GO SUB title
40 GO SUB instructions
50 GO SUB setup
60 LET start=60
70 GO SUB pick
80 GO SUB game
90 GO SUB score
100 GO SUB askmore
110 IF k$<>"n" AND k$<>"N" THEN
GO TO start
120 PRINT "End of game"
130 STOP
```

Figure 6.6 A 'core' or 'foundation' program for the example.

shall not worry about what we are going to put in for the missing bits at the moment. One important point is that we have picked names for the subroutine lines that remind us of what they are supposed to do. Once these subroutines are written, we can fill in the starter subroutine at line 9000 which assigns all of the subroutine lines to their names. Note, by the way that Spectrum doesn't allow you to use a line number greater than 9999. Get the outline right, and you'll get the program right, because details can be improved as you go along. Get the outline wrong, and not all the cleverness of Spectrum BASIC will get the thing working.

Let's get back to the program itself. As you can see, it consists of a set of subroutines that we haven't written yet. That's intentional. What we want at this point, remember, is foundations. The program follows the plan of Figure 6.5 exactly, and the only part that is not committed as part of a subroutine is the IF in line 110. What we shall do is to write a subroutine which will use INKEY$ to look for a `n` or `N` being pressed, and line 110 deals with the answer. What's the question? Why, it's the `Do you want another game` step that we planned for earlier. Take a good long look at this thirteen-line piece of program, because it's important. The use of subroutines with names in place of numbers means that we can check this program easily – there isn't much to go wrong with it. We can now decide in what order we are going to write the subroutines. The wrong order, in practically every example, is the order in which they appear. Always write the title and instructions *last*, because they are the least important to you at this stage. In any case, if you write them too early, it's odds on that you will have some bright ideas about improving the game soon enough, and you will have to write the instructions all over again. A good idea at this stage is to write a line such as:

```
111 GO TO 5Ø  ????
```

which will cause the program to skip over the title and instructions. This saves a lot of time when you are testing the program, because you don't have the delay of printing the title and instructions each time you run it. You can now write the program in sections, recording the program each time you add a new section to it. It's *always* a good idea to save programs in stages, saving under a new filename each time you add some lines. In this way, if you find that the memory of the Spectrum has become scrambled because there was a momentary power-cut, you may be able to

load one of the earlier versions of the program, and avoid having to retype all of it. Needless to say, if you use disks, you will probably save new version of the program every half-hour or so, since it takes only a second or so to save or load with disks.

The next step is to get to the keyboard (at last, at last) and enter this core program. You can't test it until the subroutine lines are written, but the next step is to record this core program and then keep adding to the core. Very often, testing takes longer than you expect, and it can be a very tedious job when you have a long program to work with. By testing each subroutine as you go, you know that you can have confidence in the earlier parts of the program, and you can concentrate on errors in the new sections. You can run a subroutine, simply by typing GO TO, then the line number of the start of the subroutine, and pressing ENTER. This will always end with an error message, but there's no harm done. You will also need to have assigned any variables that the subroutine needs. Because of this, it's always easier to test the subroutines in place, along with the program that calls them normally. Sometimes this is impossible, because the variables would be created by a subroutine that is not yet written, and it's then that you want to stick with well-tried and tested subroutines that you have used many ties before.

The subroutine routine

The next thing we have to do is to design the subroutines. Now some of these may not need much designing. Take, for example, the subroutine called 'askmore'. This is just our familiar INKEY$ routine, so we can deal with it right away. Figure 6.7 shows the form it might take. The subroutine is straightforward, and that's why we can deal with it right away! Type it in, and then add the lines:

```
9000 LET askmore=1000
9999 RETURN
```

– that form the assignments for the subroutine names. With this done, the 'askmore' subroutine is ready to operate. You can now test the core program with this subroutine in place. Type a temporary line:

```
11 GO TO 100
```

then, with no line number:

```
LET start=10
```

and you can then use `GO TO start` to test the program. You should then find that you get the screen message from the subroutine, and that typing `y` just makes the message repeat, because that's all that the program consists of at the moment. Pressing `n` will stop the program at the STOP line 130. Putting this line here prevents an error message because we would otherwise have called the subroutine incorrectly. If there

```
1000 REM Askmore
1010 PRINT : PRINT "Do you want
to try another one (y/n)?"
1020 LET k$=INKEY$: IF k$="" THE
N GO TO 1020
1030 RETURN
```

Figure 6.7 The 'askmore' subroutine, written starting at line 1000.

1. **Keep the answers as DATA lines containing ASCII codes.**
2. **Keep list of animals in another DATA line, to be placed in a string.**
3. **The number that selects the animal also selects the data line for the answer, using RESTORE.**
4. **Use variable try for number of attempts.**
5. **Use variable mark for score.**
6. **Use variable attempt to record number of attempts at one question.**

Figure 6.8 Planning the 'game' subroutine.

were no line 130, there would be nothing to stop the computer moving on from line 120 to line 1000. That would mean that the subroutine would start without having been called with a GO SUB, and when it came to the RETURN at the end of the subroutine, there would be nowhere to return to. The message you get for this kind of error is `Return without gosub`, which makes it clear, and the cure is always to put in a STOP at the end of a core program.

Now we come to what you might think is the hardest part of the job – the subroutines which are enclosed in the loop from lines 70 to 90. In fact, you don't have to learn anything new to do this. The main subroutines are designed in exactly the same way as we designed the core program. That means we have to write down what we expect them to do, and then arrange the steps that will carry out the action. If there's anything that seems to need more thought, we can relegate it to a yet another subroutine to be dealt with later.

As an example, take a look at Figure 6.8 This is a plan for the 'game' subroutine, which also includes information that we shall need for the setting-up steps in the 'pick' subroutine. Before the 'game' subroutine starts, we have to set a variable called 'attempt' to zero. This is going to be used to decide whether or not the user gets another shot at a question. We also need to pick a number at random. These steps are taken care of by a subroutine called 'pick'. This has to be separated from 'game', because if the user gets the answer wrong on the first attempt, you want to allow another try at the same animal name, not to choose another one.

The first item in the 'game' subroutine is to print the name of an animal on the screen, and ask for the name of its young. This will have been selected by the random number passed on from subroutine 'pick'. This is most easily done by keeping the names of the animals in an array. Using an array has several advantages. One of them is that it's beautifully easy to select one item at random if we do this. The other is that it also makes it easy to match the answers to the questions. If the questions are items of an array whose subscript numbers are 1 to 10, then we can place the answers in DATA lines, one set of numbers in each data line, and in the same order, so that we can use RESTORE to pick the answer.

The next point is the result of a bit of thought. We wanted, you remember, to be sure that some smart user would not cheat by looking up the answers in the DATA lines. The simplest deterrent is to make the answers in the form of ASCII codes. It won't deter the more skilled, but it will do for starters. I've decided to put one answer in each DATA line in the form of a string of ASCII codes, with the first number in each

line equal to the number of letters in the name. This way, we can RESTORE to the correct line, read the number of letters, and then start a FOR . . NEXT loop which will read that number of codes. Each code can be converted to its character equivalent, using CHR$, and added to the answer string, A$.

The other thing that the plan settles is the names that we shall use for variables. It always helps if we can use names that remind us of what the variables are supposed to represent. In this case, using `mark` for the score and `try` for the number of tries looks self-explanatory. We have used `mark`, because `score` is the name of a subroutine – you can't use a subroutine name as a name for another variable. The third one, `attempt` is the one that we shall use to count how many times one question is attempted. Finally, we decide on a name for the array that will hold the animal names – `q$`.

Playtime

Figure 6.9 shows the 'pick' subroutine put in starting at line 1050. The subroutine uses `1+INT(RND*10)` to generate a number between 1 and 10. Later, we might want to add a RANDOMIZE in line 1060 to make sure that the number was really random, but for the moment, this looks all right. How do we test it? Change line 15 to read:

```
15 GO TO 60
```

and add:

```
71 PRINT v:STOP
```

– to the program, and then RUN it several times. Each time, you should see a number appear, the value of `v`. This number might repeat values, but it should be unpredictable. If you get the same number every time in five runs, then something is wrong! Once you are sure that the number `v` is being correctly assigned, you can remove line 71, but keep the line 15 until the program is ready to run.

Figure 6.10 shows what I've ended up with for 'game' as a result of the plan in Figure 6.8. The steps are to use the random number from the 'pick' subroutine to print an animal name, and then find the answer from the user, and also the correct answer from the stored DATA lines. That's all, because the checking of the answer and the scoring is dealt with by another subroutine. Always try to split up the program as much as possible, so that you don't have to write huge chunks at a time.

```
1050 REM pick
1060 LET attempt=0
1070 LET v=1+INT (RND*10)
1080 RETURN
9000 LET askmore=1000
9010 LET pick=1050
9999 RETURN
```

Figure 6.9 The 'pick' subroutine that generates the random number.

```
1100 REM game
1110 CLS : PRINT AT 2,1;"The ani
mal is ";q$(v)
1120 PRINT AT 4,1;"The young is
called ";
1130 INPUT x$: LET try=try+1
1140 LET a$="": RESTORE 6000+v
1150 READ size: FOR j=1 TO size
1160 READ num: LET a$=a$+CHR$ (n
um)
1170 NEXT j
1180 RETURN
```

Figure 6.10 The program lines for the 'game' subroutine.

We start the subroutine at line 1100 and in line 1110 we print the name of the animal that corresponds to the random number, and ask for an answer, the young of that animal. The last section of line 1130 counts the number of tries. This is the logical place to put this step, because we want to make the count each time there is an answer. Now we set a$ to a blank. This has to be done, because if you didn't, each answer would be tacked on to the end of the previous one! By using RESTORE 6000+v, we find the DATA line that contains the correct answer. If, for example, v is 3, then the question name is in string q$(3), and the answer will be on line 6003. We read the first number in this line, size. This is the number of characters in the answer. The loop in lines 1150 to 1170 then reads in the numbers, converts them to characters with CHR$, and adds them into a$ to form the answer. That's it. All we have to do now is to compare this answer with the one (x$) that was INPUT in line 1130. This part of the work is dealt with by the 'score' subroutine. At this stage, we can't check the working of the subroutine so easily because we haven't any DATA lines to read. We could put in the DATA lines at this point, but this would mean writing the subroutine that creates the array of data. As this program consists of such short and simple subroutines, it's easy to change one if it turns out to be incorrect, so we'll move on to the 'score' subroutine.

Keeping the score

With the 'game' subroutine safely on tape, we can think now about the 'score' subroutine. As usual, it has to be planned, and Figure 6.11 shows the plan. Each time that there is a correct answer, the number variable mark will be incremented, and we can go back to the main program. More is needed if the answer does not match exactly. We need to print a message, and allow another go. When we do this, we must decrement try, otherwise it will be counted as another question, rather than as a free second shot. If the result of this next go is not correct, that's an end to the attempts. At this point, why not include some sound. We could have a different beeps for a correct answer, first mistake and second mistake. Put it all in the plan!

Figure 6.12 shows the subroutine 'score' that has been developed from this plan. It looks complicated because of the long lines, but you'll find that each line carries out

1. **For a correct answer, increment score, go to next question.**
2. **For a first incorrect answer, with** `attempt=0`**, allow another try. Decrement try, and increment attempt.**
3. **For a second incorrect answer, with** `attempt=1`**, pass to the next question, and make** `attempt=0` **again.**

Figure 6.11 Planning the 'score' subroutine.

```
1200 LET exit=1250: REM score
1210 IF x$=a$ THEN LET mark=mark
+1: PRINT AT 6,1;"Correct - your
 score is now ";
mark;" in ";try;" attempts.": BE
EP 1,12: GO TO exit
1220 IF attempt=1 THEN LET attem
pt=0: PRINT AT 6,1;"No luck - tr
y the next one."
: BEEP 5,-5: PAUSE 50: GO TO exi
t
1230 IF attempt=0 THEN PRINT AT
6,1;"Not correct - but it might
just"'"be your s
pelling. You get"'"another chanc
e - free.": BEEP 1,8: LET attemp
t=1: LET try=try
-1: PAUSE 50: GO SUB game
1240 GO TO score
1250 RETURN
```

Figure 6.12 The 'score' subroutine written. The listing shows some odd gaps because of the long lines. As usual, my printer has taken a new line after 80 characters. You should type in the words without taking new lines.

exactly one step of the plan, and that if you take it one line at a time in this way, it's really much simpler than it looks. Line 1200 contains an assignment to `exit`, the line number for the end of the subroutine. This is because if the answer is correct, we will want to move on, and if a second incorrect answer is given, we also want to move on. The more complicated case is when we have had one incorrect answer. Line 1220 then deals with a correct answer, meaning that your answer `x$` is identical to the stored answer `a$`. If it is, then the reset of this line is carried out. The variable `mark` is increased by one, and we then print the value of `mark` and the number of tries. The loudspeaker beeps (more about this in chapter 10), and the `GO TO exit` ends the subroutine by going to the RETURN line 1250. Not so bad as it looks, though my printer makes the lines look broken up because it takes a new line after 80 characters. When you type these lines, don't leave a space just before `mark` or BEEP or after the `s` of `spelling` – just type on until the true end of the line when the next item would be the next line number.

```
1300 REM setup
1310 LET try=0: LET mark=0: LET
attempt=0
1320 DIM q$(10,20)
1330 RESTORE : FOR j=1 TO 10
1340 READ q$(j): NEXT j
1350 RETURN
```

Figure 6.13 The 'setup' subroutine for dimensioning and array filling.

```
6000 DATA "Dog","Cat","Cow","Hor
se","Hen","Fox","Kangaroo","Goos
e","Lion","Pig"
6001 DATA 5,80,117,112,112,121
6002 DATA 6,75,105,116,116,101,1
10
6003 DATA 4,67,97,108,102
6004 DATA 4,70,111,97,108
6005 DATA 7,67,104,105,99,107,10
1,110
6006 DATA 3,67,117,98
6007 DATA 4,74,111,101,121
6008 DATA 7,71,111,115,108,105,1
10,103
6009 DATA 3,67,117,98
6010 DATA 6,80,105,103,108,101,1
16
```

Figure 6.14 The DATA lines with the name of the animals and the coded answers.

The next line, 1220, is only going to run if the choice is incorrect, and `a$` is not identical to `x$`. The test here is to find if `attempt=1`. If it is, the user has had one unsuccessful go at this choice before, and doesn't get another one. The 'marker' number `attempt` is set back to zero for the next try, and a message is printed. There is a (different) beep, a pause, and then its exit time again. The more awkward possibility is dealt with in line 1230. If the program reaches this line, the choice is incorrect, and we don't really need to test for the number attempt being zero, because it must be. This is the first attempt, and the answer is incorrect, so the user gets another try. The marker `attempt` is set to 1, so that another wrong answer will land you up at line 1220, and the number `try` is reduced by one because we promised that this repeat would be a free one, so we don't count this try. The second shot at the guess is done by using another `GO SUB game`, and when you have answered, the subroutine loops back because of the `GO TO score` in line 1240 to check this answer. That accounts for all of the possibilities in the plan for this subroutine. We still can't check it, because there isn't data to work with, but we can add the starting line number to the list in line 9030.

Now that we've got the bit between our teeth, we can polish off the rest of the subroutines. Figure 6.13 shows the subroutine 'setup' that deals with dimensioning

and arrays. Line 1310 sets all the variables for the scoring system to zero. Line 1320 dimensions the array q$ that will be used for the names of the animals and lines 1330 and 1340 then read the names from a data list into the array q$. We shall write the DATA lines later, as usual, so we still can't test this one.

The next step is to put in the DATA lines, Figure 6.14. There are eleven of them, one containing the names of animals, the other ten containing numbers that give the answers. The first number in each set is the number of characters to be counted, and following that we have the ASCII codes for the letters of the answer. The first letter is a capital, so that when an answer is typed, it will be rejected as incorrect unless the first letter is also a capital. Fussy things, computers! With these lines in place, we can test the whole program by altering line 15 to read:

```
15 GO TO 5Ø
```

– and then RUN the program. At this stage, the program should look as in Figure 6.15, and the questions and answers should operate correctly. The only things missing should be the title and instructions, and once we have checked that everything else is in good shape, we can type these in.

```
 10 CLS : GO SUB 9000
 11 GO TO 50
 20 REM get gosub numbers
 30 GO SUB title
 40 GO SUB instructions
 50 GO SUB setup
 60 LET start=60
 70 GO SUB pick
 80 GO SUB game
 90 GO SUB score
100 GO SUB askmOre
110 IF k$<>"n" AND k$<>"N" THEN
GO TO start
120 PRINT "End of game"
130 STOP
1000 REM Askmore
1010 PRINT : PRINT "Do you want
to try another one (y/n)?"
1020 LET k$=INKEY$: IF k$="" THE
N GO TO 1020
1030 RETURN
1050 REM pick
1060 LET attempt=0
1070 LET v=1+INT (RND*10)
1080 RETURN
1100 REM game
1110 CLS : PRINT AT 2,1;"The ani
mal is ";q$(v)
```

Figure 6.15 The complete program so far, with no title or instructions.

```
1120 PRINT AT 4,1;"The young is
called ";
1130 INPUT x$: LET try=try+1
1140 LET a$="": RESTORE 6000+v
1150 READ size: FOR j=1 TO size
1160 READ num: LET a$=a$+CHR$ (n
um)
1170 NEXT j
1180 RETURN
1200 LET exit=1250: REM score
1210 IF x$=a$ THEN LET mark=mark
+1: PRINT AT 6,1;"Correct - your
 score is now ";
mark;" in ";try;" attempts.": BE
EP 1,12: GO TO exit
1220 IF attempt=1 THEN LET attem
pt=0: PRINT AT 6,1;"No luck - tr
y the next one."
: BEEP 5,-5: PAUSE 50: GO TO exi
t
1230 IF attempt=0 THEN PRINT AT
6,1;"Not correct - but it might
just"'"be your s
pelling. You get"'"another chanc
e - free.": BEEP 1,8: LET attemp
t=1: LET try=try
-1: PAUSE 50: GO SUB game
1240 GO TO score
1250 RETURN
1300 REM setup
1310 LET try=0: LET mark=0: LET
attempt=0
1320 DIM q$(10,20)
1330 RESTORE : FOR j=1 TO 10
1340 READ q$(j): NEXT j
1350 RETURN
6000 DATA "Dog","Cat","Cow","Hor
se","Hen","Fox","Kangaroo","Goos
e","Lion","Pig"
6001 DATA 5,80,117,112,112,121
6002 DATA 6,75,105,116,116,101,1
10
6003 DATA 4,67,97,108,102
6004 DATA 4,70,111,97,108
```

Figure 6.15 *Continued*

```
6005 DATA 7,67,104,105,99,107,10
1,110
6006 DATA 3,67,117,98
6007 DATA 4,74,111,101,121
6008 DATA 7,71,111,115,108,105,1
10,103
6009 DATA 3,67,117,98
6010 DATA 6,80,105,103,108,101,1
16
9000 LET askmore=1000
9010 LET pick=1050
9020 LET game=1100
9030 LET score=1200
9040 LET setup=1300
9999 RETURN
```

Figure 6.15 *Continued*

```
1400 REM instructions
1410 CLS : PRINT AT 1,10;"INSTRU
CTIONS"
1420 PRINT : PRINT "  The comput
er will supply you"
1430 PRINT "with the name of an
animal. You"
1440 PRINT "are asked to type th
e name of"
1450 PRINT "its young. Make sure
 that your"
1460 PRINT "spelling is correct,
 and that"
1470 PRINT "you start each name
with a"
1480 PRINT "capital letter. The
computer"
1490 PRINT "will keep the score.
 You get"
1500 PRINT "two tries at each na
me."
1510 PRINT : PRINT "PRESS THE SP
ACEBAR TO START."
1520 GO SUB 1020
1530 RETURN
```

Figure 6.16 The instructions – always leave these until you have almost finished.

```
1600 REM title
1610 CLS : PRINT
1620 PRINT TAB 9;"YOUNG ANIMALS"
1630 PAUSE 300
1640 RETURN
```

Figure 6.17 The title program lines.

Figure 6.16 is the subroutine for the instructions, and Figure 6.17 is the title subroutine. The title lines include a pause, to allow time to read the title. You will now have to complete the set of GO SUB numbers in the lines 9050 and 9060, and Figure 6.18 shows what this part of the program should look like now. Now we can put it all together, and try it out. The line 15 can now be taken out, because with the title and instructions present, it isn't needed any longer. Because the program been designed in sections like this, it's easy for you to modify it. You can use different DATA, for example. You can use a lot more data – but remember to change the DIM to suit. You can make it a question-and-answer game on something entirely different, just by changing the data and the instructions. You can create much more interesting sound effects, or add some interesting graphics. One major fault of the program is that once an item has been used, it can be picked again, because that's the sort of thing that RND can cause. You can get round this by swapping the item that has been picked with the last item (unless it was the last item), and then cutting down the number that you can pick from, once you really start to flex your programmers muscles.

There's a lot, in fact, that you can do to make this program into something a lot more interesting. The reason that I have used it as an example is to show what you can design for yourself at this stage. Take this as a sort of BASIC 'construction set' to re-build any way you like. It will give you some idea of the sense of achievement that you can get from mastering your Spectrum 128. As your experience grows, you will then be able to design programs that are very much longer and more elaborate than this one by a long way. By that time, you'll know much more about the Spectrum and about programming than I could show you in the space of one book.

```
9000 LET askmore=1000
9010 LET pick=1050
9020 LET game=1100
9030 LET score=1200
9040 LET setup=1300
9050 LET instructions=1400
9060 LET title=1600
9999 RETURN
```

Figure 6.18 The lines that are needed for the GO SUB statements.

Chapter 7

Filing the data

In all computer programs, you are working with data, and this data is stored in the memory of the computer while the machine is switched on and the program is running. When you switch the machine off, both the program and the data will be lost from the memory because the memory of a small computer needs a supply of power to keep it working. Your program, however, will still be stored on the cassette or disc that you use for such purposes. The question is – what happens to the data. You see, data often takes as long to assemble and enter as does the program. If, just to take one example, you are using the Spectrum 128 to make a list of all your old Elvis singles, and their present value, all of this information may have taken hours to type in. You certainly don't want to lose it when you switch off, so what do you do about it?

One answer is illustrated in Figure 7.1. The variables that your Spectrum uses are saved along with the program that generates them and are put back into the memory when you load the program again. Now when you use the command RUN to make the program work, you automatically wipe out all of these variable values. Using RUN, in other words, means 'run from scratch'. If you start the program in any other way, the variables are still there, waiting for you. One way is to start the program with a command such as GO TO 10, which ensures that the variable values from the previous time are left unchanged. There is an automatic method, however, that makes the loading and running of such a program much simpler. This involves saving the program in such a way that it will go to the first line as soon as it has loaded, just as if you had loaded it and then typed GO TO 10, or whatever the number of the first line happens to be. The listing in Figure 7.1 has as its last line:

```
9Ø SAVE "test" LINE 1Ø
```

– which is the command that you need to record the program in this way. When this replays, then, it will start automatically as soon as it has loaded, and the start will be at line 10. All the variables that existed in the program will still exist, and the program allows you to prove just that. Type the program in and RUN it. Take the 'n' choice so that you can create new numbers, and have a cassette ready to record on, with the tape counter at zero. Now enter four numbers in the usual way – make them numbers that you can remember easily, like 11, 22, 33, 44 or some set like that. When

```
  10 REM save variables
  20 PRINT "Use old(o) or new(n)
?"
  30 LET k$=INKEY$: IF k$="" THE
N GO TO 30
  40 IF k$="n" THEN INPUT "four
numbers";a,b,c,d
  50 IF k$="o" THEN PRINT "Old n
umbers are ";a,b,c,d
  60 PRINT "Press y key to recor
d"
  70 LET k$=INKEY$: IF k$="" OR
k$=CHR$ (13) THEN GO TO 70
  80 IF k$<>"y" THEN STOP
  90 SAVE "test" LINE 10
```

Figure 7.1 A program that illustrates how variable values are saved along with a program.

you come to the recording choice, take the 'y' option, and record the program. *Don't forget to remove the 'ear' plug!* Now type `NEW` and check that the variables have been wiped out by typing `PRINT a,b,c,d` – you should get the `Variable not found` message that indicates that the variables are no longer stored. Now replay the program, using the 'Tape Loader' option if you like, or using `LOAD""`. If all is well with the tape, then your program should load, and you should see the question about using old of new variables. Type 'o' for old this time, and you should see the numbers that you originally entered appearing on the screen.

This is a very simple example, and one thing that rather spoils it is that the opening message when the program restarts is displayed on top of the listing. You can get around that problem by putting a CLS in at the start of the program, as Figure 7.2 illustrates. This one is even more mystifying if you don't know what is going on. When you have typed the program, you can't RUN it because the variables `a$`, `b` and `c$` haven't been assigned with any values. You can, however, assign these values as *direct commands*! Ignoring the program, just type something like:

```
LET a$="Look at this"
LET b=77.7
LET c$="demonstration"
```

– you can put in phrases and a number of your own, of course, and press ENTER at the end of each line. As you press ENTER on each line, the line disappears, leaving only the program on the screen. You are, however, assigning values to these three variables, and unless you clear them out by typing CLEAR (ENTER) or RUN (ENTER), these variables stay assigned. You can prove this by typing `GO TO 10`

```
10 REM save variables
20 CLS
30 PRINT a$
40 PRINT b
50 PRINT c$
```

Figure 7.2 Saving values that have been allocated by direct commands.

(ENTER), which will run the program without clearing the values. The other thing that you can do, of course, is to type SAVE"trial" LINE 10 (ENTER) and save the program on tape. When you load this in again, it will clear the screen, and print the values of the variables. This is how it's possible to save a game in the middle of playing it, and then load it in again and start from where you left off. It also means that you can record a program with a lot of variables that don't have to be read from DATA lines, because all you need to do is to type the values before you record the program. Even if you don't use the starting line number when you record, the variables are recorded with the program, and will keep their values as long as you don't use CLEAR or RUN.

You can also play this trick with arrays, but there's one change in the rules. Figure 7.3 shows a program that dimensions an array, and prints the list of values. This one will run when you try it, but there is nothing to print until values are entered. You must, however, RUN it to dimension the array. Once again, you can now enter values by direct commands, using commands such as:

LET a$(1)="Sinclair" (ENTER)

and so on. You must *not* however, make this program start at line 5. The reason is that a DIM instruction will clear memory, arranging space for variables and clearing the variables that already exist. You can make your values appear by using GO TO 10 (ENTER), and you can record the program for automatic running by using:

SAVE"newone" LINE 10

– just as before, but avoiding the line 5 that will give you trouble. You need to have run line 5 before you can enter the values into the array, but you must not use it again unless you want to create a new set. Later in this book, when we look at graphics in Chapters 8 and 9, we'll see how screen patterns can be saved and reloaded as needed. This automatic saving and reloading applies also to the SAVE! and LOAD! commands that make use of the 'silicon disc' memory. We'll deal with the use of that memory in more detail in Chapter 9.

This automatic saving of variables is very useful when the variables are used in one program only. Things get a lot more serious when you want to make use of one program to create the variables and another to use them. The assignment of variables as direct commands is a long and tedious business, and for a lot of purposes, it's useful to have a program that does this kind of job – apart from anything else, using a program ensures that the correct variable names are used. What we need for this purposes is a way of saving variables on the cassette or disk so that another program can read them in and make use of them. That sort of thing is the start of data-filing.

```
 5 DIM a$(5,20)
10 REM save variables
20 CLS
30 FOR j=1 TO 5
40 PRINT a$(j)
50 NEXT j
```

Figure 7.3 How to save arrays – you have to beware of re-dimensioning the array when the program is replayed.

What is a file?

The word 'file' occurs many times in the course of any book on serious computing. A file can mean any collection of characters which belong together. The characters of a BASIC program constitute a file, for example, because the program will not run if vital characters are missing – they belong together. A set of names and addresses in ASCII code is a file, because they form one group of information, such as our friends, or our suppliers, or debtors. A set of numbers that make up a machine-code program is a file, and a collection of the numbers that are used by a financial program is a file. In this Chapter, though, I'll take 'file' to mean a collection of information which we can record on a cassette *separately from any program.* For example, if you have a program that deals with your household accounts, you would need a file of items and money amounts. This file is the result of the action of the program, and it preserves these amounts for the next time that you use the program. More important, it allows these quantities to be used by another program, which might even use different variable names for the quantities. Taking another example, suppose that you devised a program which was intended to keep a note of your collection of vintage 78 rpm recordings. The program would require you to enter lots of information about these recordings. This information is a file, and at some stage in the program, you would have to record this file. Why ? Because if a program like this is going to be really useful, there will not be enough space even in the memory of your Spectrum computer to hold all of the information at one time. This is the topic that we're dealing with in this Chapter, recording as a separate file the information that a program uses. The shorter word is 'filing' the information.

At this point, it's important to point out some limitations. Using cassettes is fine for games that you might use for hours without needing to save or load again. Data-filing programs require a lot of saving and loading, and unless you are very patient, it just isn't on with cassettes. What follows, then, is an introduction to the principles, using cassettes so that you can learn, at no extra cost, just what is involved. If you are serious about filing, then you need at least a Microdrive, preferably a disk drive, to be able to use data-filing properly. Both of these will require new techniques and new command words, but at least if you have tried some filing techniques on cassette you'll soon learn the new techniques. Perhaps this is also a good place to point out that if you keep files about the personal details of people, you *might* need to register under the Data Protection Act of 1984. The Act was probably never intended to cover home computers, but they certainly *can* be affected.

You can't discuss filing without coming across some words that are always used in connection with filing. The most important of these words are *record* and *field* (Figure 7.4). A record is a set of facts about one item in the file. For example, if you have a file about LNER steam locomotives, one of your records might be used for each locomotive type. Within that record, you might have designers name, firebox area, working steam pressure, tractive force and anything else that's relevant. Each of these items is a 'field', an item of the group that makes up a record. Your record might, for example, be the SCOTT class 4-4-0 locomotives. Every different bit of information about the SCOTT class is a field, the whole set of fields is a record, and the SCOTT class is just one record in a file that will include the Gresley Pacifics, the 4-6-0 general purpose locos, and so on. Take another example, the file 'British Motor-bikes'. In this file, BSA is one record, AJS is another, Norton is another. In each record, you will have fields. These might be capacity, number of cylinders, bore and stroke, suspension, top speed, acceleration – and whatever else you want to take note of. Filing is fun if you like arranging things in the right order.

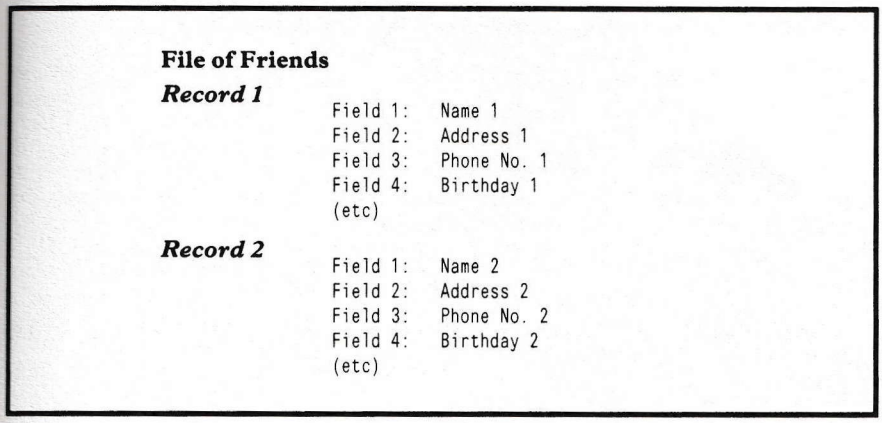


Figure 7.4 The meaning of 'record' and 'field'.

Cassette Filing

In this section, because we are dealing with the Spectrum 128 cassette system, we'll ignore filing methods that are based on DATA lines in a BASIC program, or on saving the variables along with the program. To start with, there are two types of files, only one of which we can use with a cassette system. These are serial files and random access files. The difference is a simple, but important one. A serial (or sequential) file records all the information in order on a cassette. If you want to get at one item, you have to read all of the items into the computer, and then select. There is no simple way in which you can command the system to find and read just one record or one field. A random access file does what it's name suggests – it allows you to get from the recorded data one selected record or field without reading every other one from the start of the file. Random-access filing needs a disk drive, and for that reason, we'll forget about it from now on. We'll start, then, by looking at serial files, which we can record on a cassette.

Creating a File

When you are dealing with something new, its always a good idea to start at the beginning and keep things simple at first. The first thing to establish is what can be filed. The Spectrum 128 allows you to file just one kind of data – arrays – which may be of either numbers or of strings. Don't feel that this is any kind of deprivation, because you can put any kind of information you like into an array, particularly a string array. In fact, the disk systems of other computers generally save all of their files of data in this form.

We'll start the idea of filing with the way that we make can make a recording of a string array. Figure 7.5 shows a very simple example of how a string array of data, `a$` can be recorded onto a cassette. Because the steps are simple, there's not much to look at. Lines 30 to 50 create a string array, allowing you to input whatever you want

```
10 DIM a$(5,20)
20 CLS
30 FOR j=1 TO 5
40 PRINT "Item ";j;: INPUT a$(
j)
45 PRINT a$(j)
50 NEXT j
60 SAVE "sample" DATA a$()
```

Figure 7.5 Recording a set of strings on a cassette, separate from the program.

from the keyboard. There's nothing new about this, and the important part of this listing is in line 60, in the form of:

```
SAVE "sample" DATA a$()
```

– the line that saves the data by itself rather than the program along with its data. Following SAVE we have, as usual, a filename which must not be of more than ten characters. This in turn is followed by the word DATA, used to specify that this is not a program, nor machine-code, and not a screen picture. The name of the data array that is to be saved comes next – it's `a$()`. Note that there's nothing inside the brackets, and you must not put anything inside the brackets. This last part specifies that the array called `a$` is to be saved, and to save as much of it as there is.

Now, having run this one, entered names or whatever you like into the strings, and saved, how do you get these items back? The answer is by a reading program that uses very much the same methods. You need not, of course, call the array `a$` when you read the data back into it, and in the example of Figure 7.6, we have dimensioned another array `z$`. This is the array into which the data will be read by line 10 of the program in Figure 7.6. The LOAD instruction follows the same pattern as the SAVE instruction in Figure 7.5, but this time we have `z$()` in place of `a$()`. That's all we need! In view of the limitations of using cassettes, that's as far as we go on data filing.

```
 90 DIM z$(5,20)
100 LOAD "sample" DATA z$()
110 FOR j=1 TO 5
120 PRINT z$(j)
130 NEXT j
140 STOP
```

Figure 7.6 Reading back the string file, using a FOR..NEXT loop.

Making other savings

The SAVE and LOAD commands can be used for items other than programs and arrays. From what we have used so far, you can see that using the command word DATA following a LOAD or SAVE makes the command into a load or save of an array, whose name has to be given. The Spectrum 128 also provides for saving and loading the pattern on the screen, and for the saving and loading of machine-code.

We haven't space to deal with machine code in this book, but we'll look at how it can be loaded and saved, because a lot of the programs that you buy depend on the use of machine code. We'll look first at the easier topic of saving and loading screen displays.

In general, when you work with text and numbers, saving and loading screen displays will not be of much use to you. The reason is that it's quicker and easier to use the program to create the display of words and numbers than it is to read it from tape. When you get to displays of graphics, in the following chapters, things are very different. Some displays of shapes in colour can take quite a long time to draw on the screen, and may also take up a lot of memory to program. This applies particularly to title pictures for games, and you'll see that most games packages load their title pictures from tape. In addition, there can be a lot to gain from using the spare memory, sometimes called the silicon disk for saving and loading screen pictures. Take a look, then, at the program in Figure 7.7. This generates a pattern by simple

```
  10 CLS
  20 PRINT AT 4,4;"*";AT 4,9;"**
****";AT 4,20;"********"
  30 PRINT AT 5,4;"*";AT 5,9;"*"
;AT 5,14;"*";AT 5,20;"*";AT 5,27
;"*"
  40 PRINT AT 6,4;"*";AT 6,14;"*
";AT 6,20;"*";AT 6,27;"*"
  50 PRINT AT 7,4;"*";AT 7,14;"*
";AT 7,20;"*";AT 7,27;"*"
  60 PRINT AT 8,4;"*";AT 8,14;"*
";AT 8,20;"*";AT 8,27;"*"
  70 PRINT AT 9,4;"*";AT 9,14;"*
";AT 9,20;"*";AT 9,27;"*"
  80 PRINT AT 10,4;"*";AT 10,14;
"*";AT 10,20;"********"
  90 PRINT AT 11,4;"*";AT 11,9;"
******";AT 11,20;"********"
 100 PRINT AT 12,4;"*";AT 12,9;"
*";AT 12,20;"*";AT 12,27;"*"
 110 PRINT AT 13,4;"*";AT 13,9;"
*";AT 13,20;"*";AT 13,27;"*"
 120 PRINT AT 14,4;"*";AT 14,9;"
*";AT 14,20;"*";AT 14,27;"*"
 130 PRINT AT 15,4;"*";AT 15,9;"
*";AT 15,20;"*";AT 15,27;"*"
 140 PRINT AT 16,4;"*";AT 16,9;"
*";AT 16,16;"*";AT 16,20;"*";AT
16,27;"*"
 150 PRINT AT 17,4;"*";AT 17,9;"
*******";AT 17,20;"********"
 160 SAVE "128"SCREEN$
```

Figure 7.7 A program that generates a pattern, and then saves the screen image on cassette.

methods, using PRINT AT lines to place asterisks on the screen. The last line is SAVE"128" SCREEN$, and when the program runs, you will see the pattern generated very quickly, followed by the usual message at the bottom of the screen to start the tape and press any key. Make sure that the tape counter is reset to zero, so that you can find the starting place again, and that you remember to pull out the EAR jack before recording. You can now save the picture details on the tape. This will take quite a long time, longer that you would need to save the program that created it. One bonus, however, is that saving and loading pictures like this seems to be more reliable than the saving and loading of BASIC programs. When the save action is complete, stop the recorder and wind it back to the starting point. Press the ENTER key, and you will see the pattern replaced by its listing. You can now clear the screen and load the picture, using the double command:

```
CLS:LOAD "128" SCREEN$
```

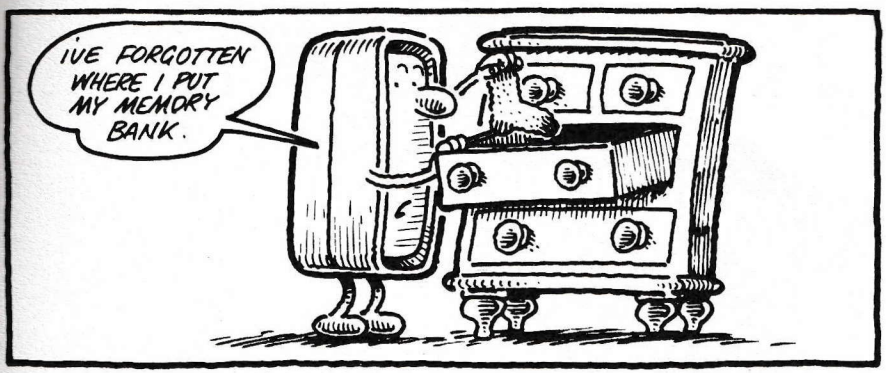
– which will clear the screen and then load in the pattern from the tape. The message Bytes: 128 will appear to remind you that this isn't a normal BASIC program, and that its name is 128. Note that there's no point in typing CLS (ENTER) and then LOAD"128" SCREEN$ (ENTER), because though the first command clears the screen, the listing will reappear whenever you start to type the second command. Only when you put both commands into one line do you get the loading on to a clear screen. As the tape runs (remember to put in the EAR jack), you will see the pattern *slowly* build up on the screen. It takes, as you would expect, as long to make the pattern reappear as it did to record it on the tape in the first place, so why would anyone use this method? Well, one reason is that the program that created a picture might be a very complicated one, and you wouldn't want to record it, then replay it to fill up a lot of the memory. Another is that you didn't want anyone who used the tape to see the program that created the picture. Another is that you might have no program for the picture. You might have created it using a light-pen, or one of the graphics-drawing programs that allows you to use the joysticks in drawing a picture. As you'll see in the following Chapters, pictures can be created in full colour, and the time that is needed to save a picture is always the same, no matter how complicated or how simple the picture may be.

There's another point to look at here, though. Try altering the last line of the program so that it reads:

```
SAVE!"128" SCREEN$
```

– and forget about the cassette recorder this time. When you run the program, the picture appears, and there's no message about tape. The picture information has been stored, however. This time it has been stored in the spare 64K of memory that the Spectrum 128 boasts of as compared to the old Spectrum 48. You can use this memory from BASIC only for saving and loading actions like this. you can't use it to write longer programs unless you are a very experienced machine-code programmer – and if you were you wouldn't be reading this book! This use of memory is often referred to as a silicon disk, because it allows you to save and load at speeds that are even faster than disks. Don't be fooled by the name, though, because this is still just memory. It may save and load fast, and without errors, just like a disk system, but when you switch the machine off, you have lost the lot, everything that was in this part of the memory as well as the program will disappear.

The usefulness of this memory is that it allows you to keep programs and screen patterns stored for instant access. Suppose, for example, that you wanted to work with two programs in turn. Instead of loading one from tape, using it, then loading the other from tape, using it, and going back to load the first one from tape, you keep



both of the programs in silicon disk. That means you load the first one, using a command like `LOAD"first"` , from the tape, and when it has loaded, you save it using `SAVE!"first"`. The exclamation (shriek) mark ! will make this SAVE to silicon disk instead of to tape, and it will be almost instantaneous, even for a long program. You can then load your second program from tape and save it to silicon disk in the same way, provided that the two programs together do not use more than 64K of memory. When you work with the programs, you can now use `LOAD!"first"` and `LOAD!"second"` as much as you like, switching from one program to the other as fast as you want to. Remember that the variables are saved with the programs, though. This cuts two ways. It means that when you load a program back, you can make it start again with all the variable values that it had when you saved it. It also means that you may need more memory to save the program in if you have altered it and created a lot of variables, so that it might not fit i the memory along with the other program. If you don't alter your programs, though, there should be no problems. Silicon disk can make it fun to use several programs, even when you have to load them in from tape to start with. If you have a disk drive, then silicon disk can be an enormous boon.

The other application of loading and saving is to machine code. Machine code consists of numbers stored into the memory, and using it requires some knowledge of how the memory is organised and used. Each byte of memory is numbered, and when you save a machine-code programs, you have to type the number of the memory byte at which the first code number is placed, and the total number of bytes that you want to save. A lot of commercial programs consist of a BASIC portion which is loaded by name, and which contains loading instructions for machine code. Sometimes this machine code may be in several sections, loading into different parts of the memory, so that you see and hear several load actions. This is why you often have to wait so long for a game to load. The SCREEN$ action is just one variation of a machine-code load and save, because the picture on the screen is obtained by storing bytes at memory locations that start at 16384, and extend for 6912 bytes. The command for saving machine code uses the word CODE, and requires the starting reference number (called the 'address' number) and the number of bytes to be saved. To save a screen picture, for example, you could use:

```
SAVE"pix" CODE 16384,6912
```

– which would have the same effect as using `SAVE"pix" SCREEN$`. The reason that it takes so long is that 6912 bytes are needed to save a complete screen. You can save and load parts of a screen, but unless you understand the rather complicated way in which the screen data of the Spectrum is arranged in the memory, this is by no means simple.

Chapter 8

Graphics I

Some prettier printing

Graphics is the word that is used to describe the use of the characters of the computer that are *shapes* rather than letters of the alphabet, number digits, or punctuation marks. Using graphics on a computer means that we can draw diagrams and pictures, and this is a feature that is not only useful for games but is valuable for presenting information in the form of pie-charts, bar-diagrams, graphs and so on. Like most modern computers, the Spectrum 128 allows you a lot of choice about the use of graphics characters. You can, for example, choose characters from a ready-made set, or you can design your own. Characters of that kind are placed on the screen using the familiar PRINT, PRINT TAB and PRINT AT commands, and because we have only 22 lines of screen and 32 characters in a line, this is called low-resolution graphics. Low-resolution often implies that there isn't much detail about the picture, but we'll see shortly that this isn't true of the Spectrum's low resolution graphics (LRG). What makes these graphics low resolution is that they can be placed only into the 22 line, 32 character positions, a total of 704 screen positions. In Chapter 9, we shall deal with high resolution graphics (HRG), which allows a much finer choice of location on the screen.

Given, then, that we are dealing in this Chapter with the use of PRINT for characters, the best place to start on our next bit of exploration of special effects is with the PRINT modifiers. As the name suggests, these cause changes on the appearance of anything that is printed on the screen. This action is the same no matter what we print – letters, digits, or graphics shapes. Take a look for starters at the program in Figure 8.1 which illustrates one of these 'effect' instructions: FLASH. FLASH is turned on when it is followed by a 1, and off when it is followed by a 0. Its effect, when it is used in a statement by itself, will be seen on anything that you print between the FLASH 1 and FLASH 0. Notice, though, that the flashing line in this program continues to flash even when the program moves on to its next

```
  10 CLS
  20 PRINT "This is an ordinary
phrase"
  30 PAUSE 150
  40 FLASH 1
  50 PRINT "This one is a flashe
r!"
  60 PAUSE 100
  70 FLASH 0
  80 PRINT "This one is not"
```

Figure 8.1 Using the FLASH attribute.

line and prints that. The flashing will continue, in fact, until the line scrolls off the screen, or the screen is cleared, or the line is replaced by another one by using PRINT AT. The instruction word FLASH is one of the *print attributes*, codes that are stored along with the character that is to be printed. These print attributes will affect the character for as long as it remains on the screen, or until they are cancelled. FLASH is a useful effect if you particularly want to bring the user's attention to something on the screen. You shouldn't overdo it, though, because a lot of flashing patterns on a screen can get too confusing. Remember, too, that flashing patterns on a screen can cause problems for anyone subject to epilepsy. The best use of a flashing message is for something really essential that you want the operator to notice, and it's often useful to combine a message like this with a sound that will make the user concentrate on the screen message. The use of an attribute like FLASH as a separate statement is called 'global' use, because FLASH 1 affects every PRINT that follows until a FLASH 0 is issued.

Another attribute that provides a set of effects which are quite different is the OVER command. The OVER command can be followed by one of two digits, 0 or 1, and Figure 8.2 illustrates the use of both OVER 1 and OVER 0. When OVER 1 has been used, any printing on the screen can be over-printed in the same colour of print. In other words, one letter can be placed on top of another, rather than replacing it, as is more usual. In this case, we are placing letters on top of spaces, because this makes the effect clearer. Now if you change line 40 to read:

```
OVER 0
```

– you will see the more normal printing action that even PRINT AT achieves. The second set of letters and spaces completely replaces the first, so that nothing that you see on the screen makes sense. Like FLASH and the other attributes, the code for

```
  10 CLS
  20 PRINT AT 4,1;"T I  I  T E
P C R M"
  30 PAUSE 200
  40 OVER 1
  50 PRINT AT 4,1;" H S  S  H  S
 E T U !"
  60 OVER 0
```

Figure 8.2 Using OVER to overprint one line with another.

```
10 CLS
20 OVER 1
30 FOR N=1 TO 5
40 INPUT K$
50 PRINT AT 11,2;K$
60 NEXT N
70 OVER 0
```

Figure 8.3 The odd effects created by printing letters on top of each other.

OVER is stored in the memory when it is used in this global way, and it acts until it is cancelled. In other words, if you have a program that contains OVER 1, and has no OVER 0 anywhere, you can find that you get the effects of over-printing when the program has ended. This can sometimes make you wonder if your sanity is all that it might be! To see what can happen, take a look at the program of Figure 8.3 in action. This allows you to enter and print up to 5 phrases. All that you type will be printed in the same place, and because OVER 1 has been used, the printed letters will be laid over each other, creating odd and often unexpected effects.

```
  10 CLS
  20 PRINT AT 4,4;"Lycee superie
ur"
  30 PRINT AT 6,4;"El lenguaje d
e computacion"
  40 OVER 1
  50 PRINT AT 4,7;"'"
  60 PRINT AT 6,26;","
  70 PRINT AT 8,10;"+ 2% toleran
ce"
  80 PRINT AT 8,10;"_"
  90 OVER 0
```

Figure 8.4 Putting OVER to useful effect.

You don't often want to make the peculiar shapes that this can provide, but for some purposes it can be useful. One example is for creating accented letters for foreign correspondence (if you write to the French or Spanish Spectrum club, for example). Figure 8.4 shows this type of use, with the OVER 1 allowing you to place the accent ' over the letter e and the , under the letter c. Another useful application illustrated here is the printing of the 'plus-or-minus' sign (±) that so often occurs in manufacturing. This is illustrated in lines 70 and 80. The underline sign has to be used rather than the genuine minus sign, because the minus sign is not in the correct position. The effect is quite good, though, and it is quite convincing on the screen. The most common use of OVER 1, however, is for underlining text, particularly titles.

Another of the PRINT attribute words is INVERSE. Using `INVERSE 1` causes anything from then on to be displayed in inverse video, meaning that black and white are reversed. As you'll see shortly when we come to deal with colours, it's not just a matter of black and white but of the background (paper) colour and the foreground (ink) colours that are reversed. Figure 8.5 shows INVERSE in use, with another

```
  10 CLS
  20 PRINT "This is normal video
"
  30 INVERSE 1
  40 PRINT "This is in inverse v
ideo"
  50 INVERSE 0
  60 PRINT : PRINT "This is in b
oth normal and ";' INVERSE 1;"in
verse"; INVERSE
0;" video"
  70 PRINT : PRINT "we can also
mix "; FLASH 1;"flashing"; FLASH
 0;" with"'" nor
mal text."
```

Figure 8.5 Illustrating INVERSE and FLASH, and showing the use of local attributes.

```
  10 CLS
  20 PRINT "This is normal brigh
tness"
  30 BRIGHT 1
  40 PRINT "This is the brighter
 one!"
  50 BRIGHT 0
  60 PRINT : PRINT "This is in b
oth normal and ";' BRIGHT 1;"bri
ghter"; BRIGHT 0
;" video"
```

Figure 8.6 The BRIGHT attribute illustrated.

```
  10 CLS
  20 FLASH 1
  30 BRIGHT 1
  40 PRINT AT 6,6;"demonstration
"
  50 BRIGHT 0
  60 FLASH 0
  70 PRINT AT 8,2;"Attr is ";ATT
R (6,6)
```

Figure 8.7 Using ATTR, and the importance of switching global attributes off in the correct order.

The ATTR number is obtained by adding up four number codes for INK, PAPER, BRIGHT and FLASH. The number codes are:

INK colour number (0 to 7)
PAPER colour number multiplied by 8
BRIGHT1=64, BRIGHTØ=Ø
FLASH1=128, FLASHØ=Ø

Figure 8.8 The ATTR numbers analysed.

important point about the way that the attribute words can be used. In the program, one line is printed in normal video and one in inverse; nothing very surprising. Line 50 cancels the inverse effect which, as usual, would last indefinitely unless you turned it off. In line 60, however, you can see normal and inverse video mixed in one PRINT line. The secret here is to have the INVERSE 1 and INVERSE Ø placed between semicolons, following the PRINT statement. This makes the INVERSE word 'local', meaning that it affects only whatever is in that PRINT statement. It does not need to be turned off unless you want the effect to end within the same PRINT statement. If you use INVERSE 1; following PRINT, then, you will not need to use INVERSE Ø if you want the whole of that line in inverse, but not the following lines of print. The INVERSE command affects everything that follows it in that line, and things look neater if the space between words is put in with the words that are in ordinary video, rather than being in inverse. Line 70 shows that we can use the same dodge with FLASH as well. Yet another of these attributes is BRIGHT. As the name suggests, using BRIGHT 1 before you print something will make the printing brighter than normal, and using BRIGHT Ø restores the normal brightness. It can produce very pleasant effects when you're using colour, but even in ordinary shades it can look impressive, as the example in Figure 8.6 shows.

Sometimes a program needs to know what is going on at a particular point in the screen – if a character is inverse, bright, flashing or likely to be printed over at that place. The Spectrum 128 provides a method of analysing this automatically, using the word ATTR. ATTR is used rather like AT, with the line and column numbers following it. Take a look at the example in Figure 8.7. In this program, both FLASH and BRIGHT are turned on, a phrase is printed, and the attributes are turned off. An important point here is that very peculiar things can happen if you don't turn off the attributes off in the correct order. This should be like loops with the last to be turned on being the first to be turned off. Since the program turns on FLASH and then BRIGHT, it must turn off BRIGHT, then FLASH. Try reversing the order of turning off, and you'll see what I mean. The point here, however, is to show ATTR in use. The numbers that follow ATTR *must* be put within brackets, unlike the numbers used with AT, but they are in the same line-then-column order. When the ATTR line runs, it prints a number on the screen, which in this example happens to be 248. Now the reasoning behind these numbers returned by using PRINT ATTR is illustrated in Figure 8.8. To analyse an ATTR number, you look to see if it is more or less than 128. If it's more than 128, then the attribute is FLASH 1, otherwise it is not. If we then subtract 128, we're left with 120. If this number is more than 64, then BRIGHT is set, and we can subtract 64 from 120 to get 56. This divides by 8 seven times with no remainder, meaning that the paper colour is 7 (white) and the ink colour is 0 (black). We'll look at these colour number in a moment, but the point here is that the value of the ATTR number analyses two of the attributes and the colours

for any position on the screen. Only FLASH and BRIGHT are detected by ATTR, INVERSE and OVER are not. If the arithmetic of ATTR looks too much, then the program in Figure 8.9 should help. This will analyse any valid ATTR number for you, showing if flash or bright has been set, and what paper and ink colours have been used.

```
  10 CLS : PRINT TAB 11;"Attribu
tes"''': PRINT "Please enter ATT
R number"''
  20 INPUT n: LET n=INT n
  30 IF n>127 THEN PRINT "Flashi
ng": LET n=n-128
  40 IF n>63 THEN PRINT "Bright"
: LET n=n-64
  50 PRINT "Paper colour is ";IN
T (n/8)
  60 PRINT "Ink colour is ";8*(n
/8-INT (n/8))
```

Figure 8.9 A program that will automatically analyse any valid ATTR number for you.

Write it in colour

It's time now to look at the colour instructions of Spectrum 128. There are three particularly important ones, which use the instruction words BORDER, PAPER and INK. We'll start with BORDER. This has to be followed by a number, the colour number for the border around the screen. The colour numbers are listed in Figure 8.10; they range from 0 to 7. Figure 8.11 demonstrates what these colours look like in the border, with a PAUSE line to prevent the colours from replacing each other too quickly to see. If you want to show a really dazzling BORDER for a short time, then try the subroutine in Figure 8.12. This uses a loop to switch from one BORDER colour to another as fast as Spectrum can obey, and it makes the border look very colourful. Note if you ever owned one of the first Spectrum models that the colours are now much improved, with no wavy patterns or other disturbances on them, even when viewed on a TV receiver.

The next items are PAPER and INK. The names tell all, really. PAPER is used to select the colour of the background, and INK to select the colour of the letters or other shapes that you print on the paper. The range of colours is 0 to 7, as before.Figure 8.13 illustrates PAPER and INK in use. An instruction such as PAPER 6 does not, by itself, cause colour to appear. If we print on the screen following a PAPER 6 instruction, the background for our printing will appear in yellow, but only for the part on which we have printed. To make PAPER 6 colour a complete screen yellow, we have to follow it with a CLS instruction. That's illustrated in line 40 of Figure 8.13. The second part of the program uses PAPER 7, white, and prints in different INK colours. You can see from the results of this program that if you want to keep your text clear, you have to use contrasting colours. Numbers that are very close to each other will never give enough contrast to make a good display. My own

Number	Colour
0	Black
1	Blue
2	Red
3	Magenta (red light + blue light)
4	Green
5	Cyan
6	Yellow
7	White

***Note:* the effect of BRIGHT is to allow you the effective use of two shades of each colour other than black.**

Figure 8.10 The colour numbers. The numbers are arranged in order of increasing brightness, as they would appear on a black-and-white receiver or monitor.

```
10 CLS
20 FOR n=0 TO 7
30 BORDER n
40 PRINT AT 11,8;"Border colou
r ";n
50 PAUSE 200
60 NEXT n
```

Figure 8.11 Displaying the BORDER colours, with PAUSE used to make a delay.

```
10 CLS
20 PRINT AT 11,8;"Border Tarta
ns"
30 GO SUB 1000
40 BORDER 1: STOP
1000 FOR j=1 TO 50
1010 FOR n=1 TO 6
1020 BORDER n
1030 NEXT n: NEXT j
1040 RETURN
```

Figure 8.12 The display of the BORDER tartan!

```
 10 CLS
 20 FOR n=0 TO 7
 30 PAPER n
 40 CLS
 50 PRINT AT 11,8; INVERSE 1; "
This is paper ";n
 60 PAUSE 100
 70 NEXT n
 80 PAPER 7: CLS
 90 FOR n=0 TO 7
100 INK n
110 PRINT "This is ink ";n
120 PAUSE 100
130 NEXT n
```

Figure 8.13 Using PAPER and INK. Note that CLS is needed following a PAPER statement if the whole screen is to be coloured.

preference is to have the PAPER colour dark, and the INK colour light, because the opposite, a white screen with black print, can appear to flicker very irritatingly. Don't expect the letters to appear in very clear colours, because colour TV sets are not very good at displaying colour in small chunks. Add to that the fact that 90% of the male population is partially colour blind, and you'll see that the most impressive colour displays are the ones that use strong colours in big areas. The default settings of PAPER and INK, meaning the settings that exist unless you change them are 0 (black) for PAPER and 7(white) for ink. These numbers are in ascending order of brightness. Note also that PAPER and INK can be used locally, as part of the PRINT line and referring only to that line.

A touch of LRG

It might sound like something fatal, but LRG is just the first letters of Low REsolution Graphics. The LRG standard characters of the Spectrum 128 can be obtained by using the keyboard directly, pressing the key marked GRAPH, and then using the number keys on the top row to get the graphics characters that are printed on these keys. This is a very much easier method than was needed in the first Spectrum models, and it makes it quite easy to assemble interesting patterns. The problem for me is that I can't demonstrate these very easily. The graphics shapes that you see on the keyboard and on the screen use ASCII codes 128 to 133, which do not print with normal printers, and do not necessarily print the same shapes even on printers that accept them. I can get around this by using CHR$ numbers, and in the examples that follow, I'll use this method in the listings. The other option, of using the COPY command, works to some extent, but can cause some rather strange-looking listings. You, however, can take the simpler option of just pressing the graphics key and the appropriate number key. The shapes and their codes are shown in the Manual on page 51. Note that if you press the CAPS SHIFT key along with a number, you get the character shown on the key, if you don't use the CAPS SHIFT you get the inverse.

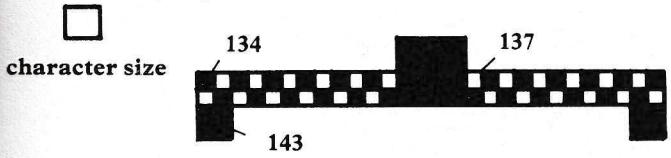


Figure 8.14 How to plan out a low-resolution graphics shape on graph paper.

The next step is how to make these graphics shapes produce something interesting on the screen. You can, in fact, compose simple drawings on to the screen directly, using the graphics key and the number keys, with or without the CAPS SHIFT. Since you can delete any part of the pattern, drawing in this way is quite surprisingly easy, but it's not always easy when you have more than one line of drawing to get everything into the correct position. A method of planning that is very useful, no matter how you actually enter the graphics shapes, is shown in Figure 8.14. This uses graph paper to draw the shape that you want in pencil, and then fill in the actual graphics shapes in ball-pen. You can then identify the ASCII code numbers for these shapes so as to put them into a program. This has been done in Figure 8.15 for the simple 'flying saucer' shape drawn in Figure 8.14, using the CHR$ numbers. Working direct from the keyboard, this would be considerably easier. Another thing that can make these graphics useful is to assign a set of graphics shapes to a string. You can, for example, type LET G$="... and then press the GRAPH key and type shapes instead of letters. It's easy enough for one line, but it's not so easy if you want to put more shapes into the next line down. You have to remember also to press the GRAPH key again before trying to use the other keys. The advantage of using one string name for a graphics shape is that you can then print the shape at various parts of the screen, using PRINT AT, and even animate it in a rough way. Figure 8.16 shows a falling shape, animated by printing it in a position, waiting briefly, and then wiping the pattern by printing a blank space in the same place. The same pause is then repeated, and the next space down is selected for the next printing. This kind of animation reveals why these graphics are referred to as being of low resolution, because the movement always looks jerky. This, in turn, is because there is such a large movement from one position to the next.

```
  10 CLS : LET s$=""
  20 FOR j=1 TO 14: READ d: LET
s$=s$ +CHR$ (d): NEXT j
  30 PRINT AT 6,15;CHR$ (143);CH
R$ (143)
  40 PRINT AT 7,9;s$
  50 PRINT AT 8,9;CHR$ 143;AT 8,
22;CHR$ 143
  60 DATA 134,134,134,134,134,13
4,143,143,137,137,137,137,137,13
7
```

Figure 8.15 A program that produces the shape shown in Figure 8.14.

```
10 CLS
20 LET g$=CHR$ 134 +CHR$ 140+C
HR$ 137
30 FOR n=0 TO 21
40 PRINT AT n,14;g$
50 FOR j=1 TO 5: NEXT j
60 PRINT AT n,14;"    "
65 FOR j=1 TO 5: NEXT j
70 NEXT n
```

Figure 8.16 Animating the shape by drawing, waiting, wiping, waiting then repeating this process at a new position.

User-defined graphics

User-defined graphics means creating your own graphics characters, which allows you to make much more interesting shapes that you can get from the keyboard. In addition, a user-designed graphics shape can contain a lot more detail, and can look like a piece of high-resolution drawing. Like any other character shape, however, it can be printed only in the preset 22-line, 32 column positions, so that any animation will as before, look rather jerky. In addition, creating user-defined shapes requires quite a lot of both planning and programming. A total of 21 of these user-defined shapes can be created, and you can use them in two different ways. One direct way is to press the graphics key followed by a letter key, because each graphics shape that you make is assigned to one of the letter keys, but you don't get the graphics shape unless you press the GRAPH key before pressing the letter key. The other standard method is to use the CHR$ numbers, which range from 144 to 164. In this book, since the shapes don't print easily on my printer, I'll stick with the CHR$ method.

To start with, you have to plan out the shape of each of your characters that you want. The shapes are drawn on a set of 8 × 8 squares, such as is shown in Figure 8.17.

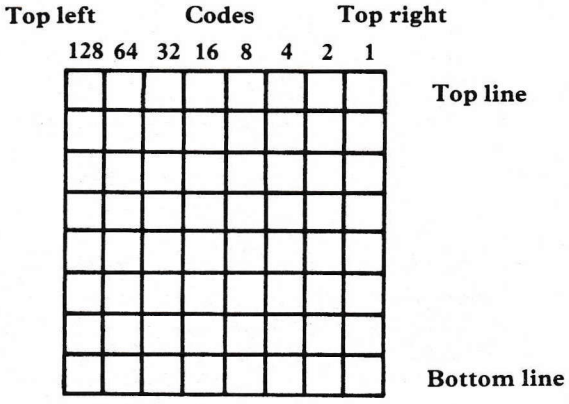


Figure 8.17 The planner for your own graphics characters.

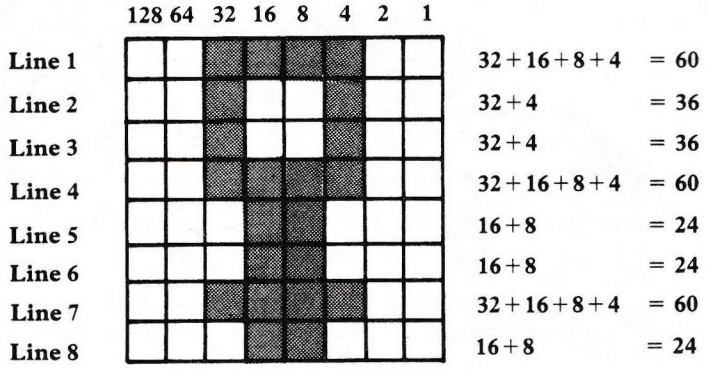


Figure 8.18 Illustrating the use of the character planner with a symbol.

This gives a total of 64 squares from which each single character shape can be made up, and you need to remember just how small the shape will appear to be when you see it on the screen. You draw your character shape by shading in squares – you can't have half a square shaded, it's full squares or nothing. An example, a symbol for a program on biology, is shown in Figure 8.18. The squares that are shaded in on this diagram will, in fact, be the INK colour squares when the character is displayed. The next step after filling in the shape is to write down the code numbers for each line. At the top of the diagram, the numbers show the code for each position in the line, with the left-hand square having value 128, and the right-hand square 1. Write down the position number for each shaded square in a line, and add these numbers up. If there are no shaded squares in a line, the total will be zero, if all the squares are shaded, the total will be 255. After doing this, you will have a set of eight numbers. If you want to fit this character into a line with others, you will need to leave a space at the top and the bottom – this has not been done in this example.

Now the number codes for the shape have to be programmed into the memory of the computer. Placing numbers in the memory is done by using the command word POKE, but there is a special version of the instruction for this purpose. First of all you pick which letter key will be used (with the GRAPH) key to produce this shape. Suppose, for example, that you decide to use the letter 'a'. The Spectrum 128 locates the correct part of its memory for the new character by the phrase `USR"a"`. The eight numbers are placed into eight consecutive positions in the memory starting at the USR"a" address, so that we use `USR"a"`, `USR"a"+1`, `USR"a"+2` and so on. In fact, the whole thing can be done with a FOR . . NEXT loop, as the listing in Figure 8.19

```
  10 CLS
  20 FOR j=0 TO 7: READ d: POKE
USR "a"+j,d: NEXT j
  40 PRINT AT 6,15;CHR$ 144
 100 DATA 60,36,36,60,24,24,60,2
4
```

Figure 8.19 How the numbers that are obtained from the planner are used to create the character shape.

shows. The numbers that have been used in the DATA line here are the numbers from the drawing of Figure 8.18, reading from the top line downwards. You must normally use eight numbers – it's not enough to read in only the lines that have shaded squares. You can, however, with a bit of cunning, use this method to make changes to the letters whose shapes are normally stored in these addresses. These are the normal upper-case letter of the alphabet, A, B, C, and so on. If you poke new numbers into part of an address, what you will get will be partly the letter that is normally stored there and partly your own new shape. To see what numbers are already stored in these addresses you need to know where to look. The clue is to start at address number 65368. For example, if you use the command:

```
FOR n=65368 TO 65375:PRINT PEEK(n);" ";:NEXT n
```

– and press ENTER, you will see the numbers for your GRAPHICS A appear. The next set of eight numbers will be for "B", the next for "C" and so on.

Finally, because these changes in codes are numbers stored in the memory, you can save them and load them. You can save an individual character by using a command such as:

```
SAVE"fem" CODE USR"a",8
```

– but this can be very long-winded when you want to save a large number of these shapes. A better way is to save the whole piece of memory that they use, by using:

```
SAVE"shapes" CODE 65368,16Ø
```

– because this when it is loaded back will pack the correct part of memory with the codes. When this is to be loaded back, you need only the command:

```
LOAD"shapes"CODE
```

– which will ensure that the codes go back into the memory address from which they were saved. This way, you get back your special shapes. This also avoids the need to have the program that creates the shapes in with the program that uses them. Once again, this is a technique that a lot of the games programs use.

Chapter 9

Graphics II

High resolution

The high-resolution graphics abilities of the Spectrum 128 are excellent, but in order to make use of them you need to be familiar with the commands that are used, and how they operate. These commands (we should really call them 'statements' when they form part of a program) are very different from the ones such as PRINT that we use with text. One concept that we need to take a look at first of all is *resolution*. The resolution of a graphics display means the number of distinct pieces that would be needed to fill the screen. If you look closely at the screen of a colour TV, you will see that it is divided into a set of lines or dots. These are the bits that glow and give out light, and you can't have anything displayed on the screen which is smaller than one screen dot or the width of a line. In fact, unless you use a monitor, you can never get anywhere near displaying dots so small or lines so narrow. The 'dots' that the computer can work with are called 'pixels' (shortened from **pi**cture **el**ements). Each pixel that a computer can control corresponds to the size of several dots on the TV screen. The Spectrum 128 allows you to use the characters blocks as pixels for low-resolution patterns. This gives you the use of $32 \times 22 = 704$ positions on the screen, but you can also make use of high-resolution graphics with 45056 screen positions! Not only can you make drawings that use these much closer-spaced pixels, you can also mix drawings with ordinary text and with user-defined characters. This is something that is not so easily achieved in other computers, and it helps to explain why Spectrum is still the best-selling computer for home use.

The next step, then, is to look at the instructions that will work with these individual pixels. To do so, we need a method of specifying the position on the screen that we want to work with. For text, we could use the line and column numbers with PRINT AT. The line numbers were 0 to 21, and the column numbers were 0 to 31, so that we could specify any character position with just two numbers. For high resolution, we also use two numbers, but the sizes are considerably greater. The column number can now be in the range 0 to 255. Like the text column numbers, position 0 is at the left-hand side of the screen, and 255 is at the right. The line number range is 0 to 175. Unlike the text line number, position 0 is at the bottom of the screen and 275 at

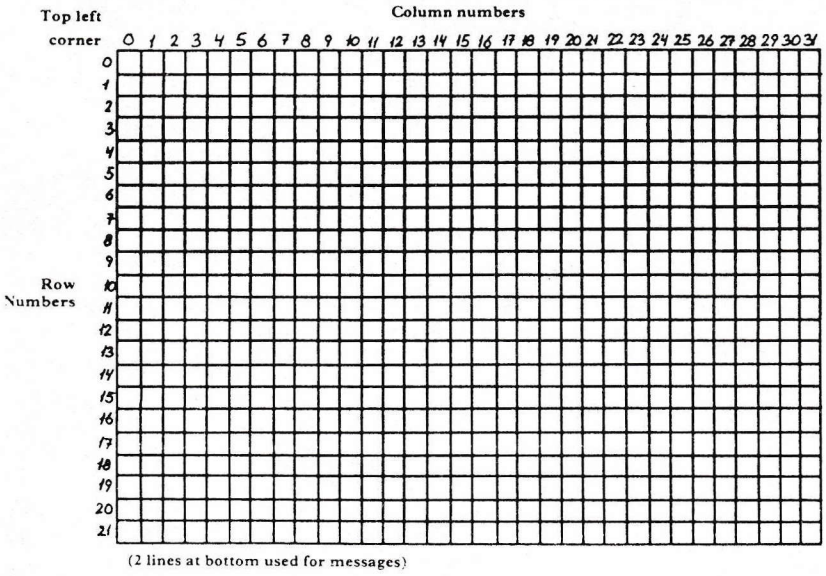


Figure 9.1 How you can construct your own high-resolution graphics map, using graph paper.

the top, so that the point whose line and column numbers are both zero is located at the bottom left-hand side of the screen. Figure 9.1 shows the 'map' of numbers for the high-resolution screen. When we make use of these numbers, we always place the column number first, separated by a comma from the line number. These numbers correspond so closely to the coordinate system that is used for plotting graphs that we often use the same names. We call the column number the 'x coordinate', and the line number the 'y coordinate'.

A new plot

The keyword PLOT is one that is used to light up one single pixel. PLOT is followed by the x and y coordinate numbers, separated from each other by a comma. You might, for example use `PLOT 127,87` to light up the pixel at the centre of the screen. The colour that you should see will be whatever INK colour you are using. You can put an INK colour locally into the PLOT by using something like `PLOT INK 1;127,87`. Don't expect the colour to be very accurate if you are using a TV receiver to display the picture. It's likely, in fact, that anything you plot, using a TV receiver to display the picture, will be surrounded with other colours, so that you might not notice a change in colour of the point you are plotting. As I said earlier, colour is really effective only when it exists in fairly large areas. In addition, there are some rules to learn about the way that you can use colour in the high-resolution instructions. The main restriction is that you need to know what the effect of the colour of one pixel will have on another. In general, each *character position* can have only one INK colour and one PAPER colour. Note I said character position, meaning a set of 8 × 8 pixels in a block.

```
10 PAPER 1: CLS
20 PLOT INK 6;127,87
30 PAUSE 100
40 PLOT PAPER 2;127,87
50 PAUSE 100
60 PLOT INK 5;126,85
```

Figure 9.2 Plotting pixels. These are very small, and colour never looks correct in such small dots.

It's not easy to illustrate these effects on single pixels, as Figure 9.2 shows, because the pixels are small, and they are not well displayed on a TV receiver. If you are using a monitor, however, you will be able to see the effect much more clearly. In this example, a paper colour is selected and used to paint the whole screen because it is followed by CLS. A pixel is then plotted, and can be seen at the centre of the screen. After a short pause, there is another PLOT, but this time to set the PAPER colour in the same spot. As you can see, its effect is to set a whole character square to the paper colour of red that has been selected. When another pixel is plotted in a different INK colour, you can see the other dot appear, but on a TV receiver you can't really tell if the colours are the same or not.

```
10 PAPER 1: CLS
20 FOR y=88 TO 91
30 FOR x=128 TO 135
40 PLOT INK 3;x,y
50 NEXT x: NEXT y
60 PAUSE 200
70 FOR y=92 TO 95
80 FOR x=128 TO 135
90 PLOT INK 6;x,y
100 NEXT x: NEXT y
```

Figure 9.3 Illustrating how the colour of one set of pixels in a block overrules another set made earlier.

Figure 9.3 makes it all rather clearer. In this example, a loop is used to colour in half of the pixels in a character block in one colour, and there is a pause so that you can see the effect. The other half is then plotted in a different colour – but the moment that this action starts you can see the first half change colour. All of the 64 pixels in the block end up at colour 6 because only one INK colour is allowed in each group, and it will be the latest one used. This means that your high-resolution drawings must use low-resolution colour, and the restriction is because of the amount of memory that would be needed to allow each pixel to be displayed in its own colour. The important thing to look at is how this restriction affects the attributes at each pixel point. Using OVER 1 or INVERSE 1 has the same effect on a point plotted in INK colour – to make the colour the same as the PAPER colour. In other words, this makes the point invisible, as Figure 9.4 illustrates. Remember, as always, that these effects have to be turned off if they are used in their global form.

The effects to watch are BRIGHT and FLASH, as Figure 9.5 demonstrates. The program sets the paper colour to 1, blue, and the ink to 6, yellow. Plotting with FLASH 1 makes the dot flash, but the whole of the character square flashes also. This is

```
10 PAPER 1: CLS
20 INK 6
30 PLOT 127,87
40 PAUSE 150
50 PLOT OVER 1;127,87
60 PAUSE 100
70 PLOT OVER 0;127,87
80 PAUSE 150
90 PLOT INVERSE 1;127,87
100 INVERSE 0
```

Figure 9.4 The action of OVER1 and INVERSE1 when used with PLOT.

```
10 PAPER 1: CLS
20 INK 6
30 PLOT FLASH 1;127,87
40 PAUSE 100
50 PLOT BRIGHT 1;100,150
60 PAUSE 100
70 BRIGHT 0: FLASH 0
```

Figure 9.5 Showing the effect of BRIGHT and FLASH along with PLOT.

```
10 PAPER 0: CLS
20 INK 6
30 CIRCLE 50,120,40
40 PAUSE 100
50 CIRCLE PAPER 4;200,50,40
```

Figure 9.6 A simple circle-drawing program, using CIRCLE.

```
10 PAPER 0: CLS
20 INK 6
30 CIRCLE 50,120,40
40 PAUSE 100
50 FOR y=11 TO 91 STEP 8
60 FOR x=155 TO 243 STEP 8
70 PLOT PAPER 3; INVERSE 1;x,y
80 NEXT x: NEXT y
90 CIRCLE 200,50,40
```

Figure 9.7 Putting the circle in a 'window' of colour.

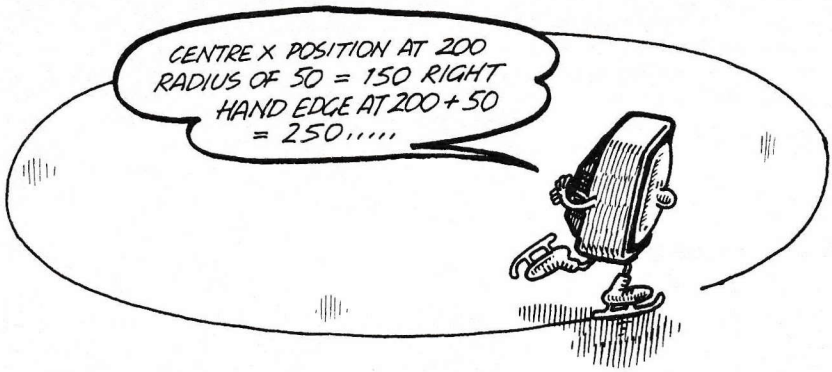
seldom what you want to happen. When `BRIGHT 1` is used, it is the PAPER colour that is bright, in this case a brighter blue, not the INK colour. Again, this might not be exactly what you expected. In all of the high-resolution drawing instructions, then, you have to use attribute instructions like FLASH and BRIGHT with care. If you keep to one PAPER colour and one INK colour for the whole screen, then you will have no difficulty, but you have to avoid FLASH and BRIGHT when you are working with several different colours in different parts of the screen.

Drawing the line

The action of PLOT is to move a *graphics cursor*, which unlike the ordinary cursor that you see on the text screen is very small (one pixel) and not always visible. If you PLOT in INK colour, you see one dot appearing as a cursor to show the position of the plot, but if you plot in PAPER colour, the dot is invisible. Now it would be very difficult, and it would require an enormous amount of programming if you needed to PLOT each separate point in a drawing, and it's fortunate that you don't need to. The Spectrum 128 is provided with two particularly useful instruction words for drawing shapes, and though it seem more logical to start with drawing straight lines, the easier of these instructions deals with circles, so that's where we'll start.

The CIRCLE instruction allows you, as the name suggests, to draw circles – and only circles. This might seem obvious, but the other drawing instruction of the Spectrum 128 can also draw circles and a variety of other shapes, whereas the CIRCLE statement does just the one job. When you use CIRCLE, you have to specify three quantities. These are the `x` and `y` numbers for the position of the centre of the circle, and the radius of the circle, which is the distance from the centre to the rim. The numbers that you use are the pixel-count numbers that should be familiar by now, 0 to 255 in the horizontal (x) direction, and 0 to 175 in the vertical (y) direction. It's easy enough to position the centre of the circle, but what you need to beware of is setting impossible values of radius, so that the circle cannot be completely drawn.

Figure 9.6 shows a simple circle-drawing program. Line 10 sets the main paper colour and clears the whole screen to this colour. Line 20 sets the INK colour, and line 30 draws a circle at the top left-hand corner of the screen. This circle should appear as yellow on black, but because of the limitations of TV receivers you will probably see other colours on the rim of the circle. If you are using a colour monitor, however, the colours should be fairly good. After a pause, another circle is drawn. This uses a PAPER instruction as part of the CIRCLE instruction, so that the paper colour around the rim of the circle will be changed. As you can see when this runs, it produces a circle with boxes of the new PAPER colour around it. The important point is that for a circle of this size, you don't get the inside of the circle filled with colour, nor do you get the circle appearing inside a neat box in the new PAPER colour. You can, however, make a PAPER box for a circle by using a loop, remembering that you only need to plot the PAPER colour of one pixel in each 8×8 box to fill in the whole box. Figure 9.7 shows the lines that you have to add so that the second circle is in a 'window' of a different PAPER colour. It takes some time to draw in the PAPER colour, but the effect is good. The numbers that have been used consist of steps of 8, starting outside the CIRCLE dimensions. The edges of the circle extend to points whose position numbers are found by adding or subtracting the radius number to/from the x and y numbers. For example, the left-hand side of



the circle is the centre x position of 200 minus the radius of 50, giving 150, and the right-hand edge is at 200 + 50 = 250. The pixels at the centres of the character squares are at x positions 3, 11, 19 . . . and so on, going 8 steps across each time. The nearest centre is at 155, so that's where we start plotting the x values. We go in steps of 8 to 243, the nearest pixel centre to the right-hand edge. The y numbers are treated in the same way, starting at 11 and going to 91. The PLOT line uses both PAPER and INVERSE 1, because we want to put in the new PAPER colour here, but we don't want the points plotted in INK colour. If you omit the INVERSE 1 here, you'll see the INK dots plotted, which is not what we want.

Having introduced the CIRCLE statement, then, what can we do with it. The obvious point is that it allows us to draw circles any where we like on the screen. It's not all straightforward, though. If at any point in tracing a circle, a point would be outside the screen area, then your program will stop with an Integer out of range message. You must make sure, therefore that this can't possibly happen. It's easy enough when you know the numbers in a CIRCLE statement, because you can check by adding and subtracting the radius number to and from the x and y numbers. If any of these goes outside the range 0–255 for x or 0–175 for y, then you will get the out of range message. For example, if you program CIRCLE 1ØØ,8Ø,9Ø, then because 80–90 = –10, you will get the error message when the lower part of the circle is being drawn. The error is found only when the circle is actually being drawn. If the numbers are represented by variables and are outside your control (entered from the keyboard, perhaps), then you need a mugtrap line to check them before they get to the CIRCLE statement.

There are two elaborations to drawing circles that can be useful. One of them is to fill the circle with INK colour. There's no colour filling statement for the Spectrum 128, unlike so many of today's computers, so that the job of filling a circle with colour has to be done with a lot of plotting or drawing. The other action that is sometimes useful is drawing circles as part of a loop. This allows things like concentric circles and tubes. Take a look, for example, at Figure 9.8, which deals with a number of useful topics. It starts with setting PAPER and INK colours, and then draws a spider-web pattern of concentric circles, meaning circles that all have the same centre. Notice that line 30 assigns values to x and y even though these are not changed in the course

```
 10 PAPER 1: CLS
 20 INK 6
 30 LET x=127: LET y=87
 40 FOR r=4 TO 80 STEP 2
 50 CIRCLE x,y,r
 60 NEXT r
 70 SAVE !"circ"SCREEN$
 80 CLS
 90 LET x=45: LET y=45
100 FOR r=40 TO 20 STEP -1
110 CIRCLE x,y,r
120 LET x=x+4: LET y=y+4
130 NEXT r
140 SAVE !"tube"SCREEN$
150 FOR n=1 TO 10
160 LOAD !"circ"SCREEN$
170 PAUSE 25
180 LOAD !"tube"SCREEN$
190 PAUSE 25
200 NEXT n
210 ERASE !"circ": ERASE !"tube
"
```

Figure 9.8 Drawing patterns with circles, and saving on the 'silicon disk'.

of the program. The reason is that once an assignment has been made, variables can be dealt with rather faster in a program than numbers like 127 or 87. Even with this precaution, the circles are drawn slowly, and a clue to how to get instant circles is shown in line 70. This saves the whole screen, just as it is, to the silicon disk memory. Lines 90 to 130 then draw a tube. This is done by picking a starting point and a circle radius and then drawing a set of circles whose starting point is shifted for each circle and whose radius is made one step less. This is quite effective, and it's good to watch in action. This also is saved to silicon disk, using the SAVE! statement. Now comes the crunch. You can get round the slow drawing of these shapes, once they have been drawn once, by loading them from the silicon disk memory. The loop in lines 160 to 200 shows this being done, and you can see that one pattern can be replaced by another almost instantly.

The silicon disk allows very convincing animation to be carried out, because the screens can be loaded very fast, and all you need to do is to have a set of patterns read and stored in the memory. There's a limit, however, to how many screen-patterns you can store – the absolute limit is 11. You must also make sure that there are no patterns stored there before you start. If you run the program of Figure 9.8, for example, without the line 210, you'll find that the program will run only once. This is because when the next SAVE! statement is run, the machine finds that there is a screen file of the same name already stored, and will stop with an error message 'File already exists'. The files in the silicon disk memory remain in store even after a NEW has removed a program, so that the only way to clear this memory of files, short of switching off or resetting, is to use the ERASE statement. You can, of course, also use this as a direct command before you start a program running. You can find what files exist in the silicon disk memory by typing CAT! (ENTER), which will give a list of the filenames.

```
 10 PAPER 1: CLS
 20 INK 6
 30 LET x=45: LET y=45
 40 FOR r=40 TO 30 STEP -1
 50 CIRCLE x,y,r
 60 LET s$=CHR$ (r+35)
 70 SAVE ! s$SCREEN$
 80 LET x=x+8: LET y=y+8
 90 CLS
100 NEXT r
110 PAUSE 200
120 FOR n=40 TO 30 STEP -1
130 LET s$=CHR$ (n+35)
140 LOAD ! s$SCREEN$
150 NEXT n
```

Figure 9.9 Animation using the silicon disk to store the patterns. Note how the pattern number has been used as a filename.

Figure 9.9 shows animation of this type. The starting program draws circles, and each circle pattern is stored. After a pause, the pattern is replayed, and you can see how much faster this is, even with a PAUSE included, than the original drawing. For good animation, you need a smaller amount of movement between screens, but the principle is sound. As before, once this program has run, you can only run the first part again if you erase all of the stored files. If you do this in the second part of the program, by having a line such as :

```
145 ERASE! s$
```

– you will find that the program runs very slowly, because a pattern takes as long to erase as it did to draw in the first place. The main snag with this method is that you have to wait to see the patterns drawn before you can use them. You might think that you can use PAPER of the same colour as the INK when you create the drawing, and then use a different PAPER colour when you LOAD! the image back, but this doesn't work because the original paper colour is loaded back also. The best way out of it is the difficult way – to load the screen images direct from tape or disk into the silicon disk memory – but that sort of programming is well beyond the scope of this book!

Lay it on the line

The other drawing instruction word of the Spectrum 128 is DRAW. Unlike CIRCLE, DRAW is a multi-purpose instruction, and it can be used in a lot of interesting ways. We'll start this section by looking at the simplest uses of DRAW, and then move progressively to the more complicated ones. The simplest possible DRAW statement is `DRAW x,y,` where `x` and `y` are the usual position numbers. What this produces depends on what you have been doing up to then, as Figure 9.10 shows. In this example, the first line is drawn from the bottom left-hand corner of the screen to the point in the middle that is specified by 127,87. In other words, if you haven't used a PLOT or DRAW or CIRCLE that placed the graphics cursor somewhere on the screen, the line that DRAW produces will start from point 0,0, the bottom left-

```
10 PAPER 1: CLS
15 INK 6
20 DRAW 127,87
30 PAUSE 100
40 DRAW 100,50
```

Figure 9.10 An example of DRAW in action, showing that the position numbers are *relative*.

hand corner. Once a position has been used, however, the DRAW starts from that point. The next bit is not so easy. As you will see, the instruction in line 40 extends the line at a different angle but still going from left to right across the screen. It is *not* a line drawn from 127,87 back to 100,50. What happens is that the line is drawn 100 pixels right and 50 pixels up from where it started, which was 127,87. It therefore ends up at 227,137, not at 100,50. The effect of a DRAW action, then depends on where you start from. This might look confusing but, as you'll see, it can be very useful indeed.

Take, for example, the program of Figure 9.11. This draws a set of squares of different sizes at different places – but the square-drawing is done by a single subroutine. This is the advantage of this type of drawing, because once you have a subroutine to draw a shape, the same subroutine will do for any size and position of this shape. This wouldn't be possible if the DRAW action used screen-position numbers, and it's only because DRAW uses what are called *relative* position numbers that we can do this. The essential part of the program here is the square subroutine, and it uses variable s, the size of the side of the square. The square is drawn one side at a time, and by using DRAW s,0, you force the drawing to give a line from the cursor position to a point s pixels to the right. There is no change in the y number, so the top of the square remains level. The next DRAW is 0,-s, so that the horizontal position stays as it was, but the vertical side is drawn, going s pixels *down*. The negative sign is what specifies that the direction is down rather then up. In the next DRAW, using -s,0 causes the line to be drawn to the left, and the final 0,s is the last side, going upwards the join the original position.

An essential part of the main program is the PLOT and if you don't believe it, try missing it out. Unless PLOT is used, the program has no starting point, and it will take the point 0,0 as a start. This makes it impossible to move downwards, so you get

```
10 PAPER 1: CLS
20 INK 6
30 LET x=50: LET y=50
40 FOR s=40 TO 20 STEP -2
50 PLOT x,y
60 GO SUB 100
70 LET x=x+4: LET y=y+4
80 NEXT s
90 STOP
100 DRAW s,0: DRAW 0,-s
110 DRAW -s,0: DRAW 0,s
120 RETURN
```

Figure 9.11 Taking advantage of relative drawing numbers to write a square subroutine. This allows squares to be drawn anywhere on the screen in any size, providing that the line never goes off-screen.

```
10 PAPER 1: CLS
20 INK 6
30 PLOT 75,47
40 DRAW 100,0,1
50 DRAW 0,80,1
60 DRAW -100,0,1
70 DRAW 0,-80,1
```

Figure 9.12 Adding a third number to the DRAW instruction to make the line into a curve.

```
10 PAPER 1: CLS
20 INK 6
30 PLOT 75,37
40 DRAW 100,0
50 DRAW 0,80
60 DRAW -100,0,PI
70 DRAW 0,-80
```

Figure 9.13 The effect of using *PI* as the third number of one side.

```
10 PAPER 1: CLS
20 INK 6
30 LET x= 75: LET y=37
40 FOR s=20 TO 40
50 PLOT x,y
60 GO SUB 100
70 LET x=x+4: LET y=y+4
80 NEXT s
90 STOP
100 DRAW s,0,PI
110 DRAW 0,-s,PI
120 DRAW -s,0,PI
130 DRAW 0,s,PI
140 RETURN
```

Figure 9.14 *PI* used for four sides of a square. Contrast with the use of *–PI*.

the 'Integer out of range' message. If you put in a PLOT x,y as a line 35 instead of having it as line 50, then you will see the starting point for each square remain in the same place. When you use these DRAW subroutines, then, you have to make certain that you have used a PLOT or similar statement to place the starting-point where you want it before you call the subroutine. The simplest subroutines are, of course, for simple geometrical figures like squares, but more complicated patterns are possible. By using variables for the movement numbers in drawing subroutines, you can draw the pattern in any place and of any size that will fit on the screen. This makes it unnecessary to have the SCALE instructions that some computers need.

That isn't the end of DRAW, though. Obviously, all the OVER, INVERSE, FLASH and BRIGHT attributes work with DRAW as they do with other statements in this group, but the most interesting thing about DRAW is that you can add another number to the two that specify the movement. The third number is an angle number, and it allows you to draw patterns that have round sides rather than flat sides. Just how round depends on the number that is used, and Figure 9.12 illustrates what you get by using 1 as this number. The shape would be a rectangle if this third number had not been added, but the addition of the 1 has bowed out each side, making it look barrel-shaped. You don't have to use 1 or any *whole* number, and you can make the curvature less by using numbers like .5 or less. If the number is negative, like –.8, then the curvature is in the other direction, inwards. Using the number PI produces a circle. This is illustrated in Figure 9.13, in which three sides are straight, but the top is a circle. Once again, using –PI should produce an inward-bulging circle, but as Figure 9.14 shows you can get inward-curving flower shapes by using PI, and changing this to –PI produces outward curves! The moral with PI is try it and see!

Chapter 10

Sounds unlimited

The BEEP

The ability to produce sound is an essential feature of all modern ccomputers. The sound of the Spectrum 128 computer comes from the loudspeaker of the TV receiver that you use to see the display, so you have more control over the volume of this sound than is possible with a lot of other computers. You can also use the sound if you have a monitor with an amplifier and loudspeaker, or if you connect the output from the Spectrum to an amplifier system. To use either a monitor or an amplifier, however, you will need to have connecting leads made up specially. You can record the sound on a cassette recorder, using the leads that you normally use for recording programs – but remember that you must keep the EAR plug out when you are replaying the sound. In addition to these different methods of using the sound signal, the Spectrum 128 computer allows you a number of different ways of creating sound effects, depending on whether you want just a reminder, a melody, or a pistol shot.

What we call sound is the result of rapid changes of the pressure of the air round our ears. Everything that generates a sound does so by altering the air pressure, and Figure 10.1 shows how the skin of a drum does this. All other musical instruments also rely on the principle of something that vibrates, and pushes the air around. Air pressure, however, is invisible, and we don't notice these pressure changes unless they are fairly fast, and we measure the rate in terms of cycles per second, or Hertz. A cycle of any wave is a set of changes, first in one direction, then in the other and back to normal, which we can illustrate by the graph in Figure 10.2. The reason that we talk about a sound 'wave' is because the shape of this graph is a wave shape.

The frequency of sound is its number of Hertz – the number of cycles of changing air pressure per second. If this amount is less than about 20 Hertz, we simply can't hear it, though it can still have disturbing effects. We can hear the effect of pressure waves in the air at frequencies above 20 Hertz, going up to about 15000 Hertz. The frequency of the waves corresponds to what we sense as the 'pitch' of a note. A low frequency of 80 to 120 Hertz corresponds to a very low-pitch bass note. A frequency of 400 or above corresponds to a high pitch treble note. Human ears are not sensitive to sounds whose frequency is above 20,000 Hertz (20 kilohertz), but many animals can hear sounds in this range.

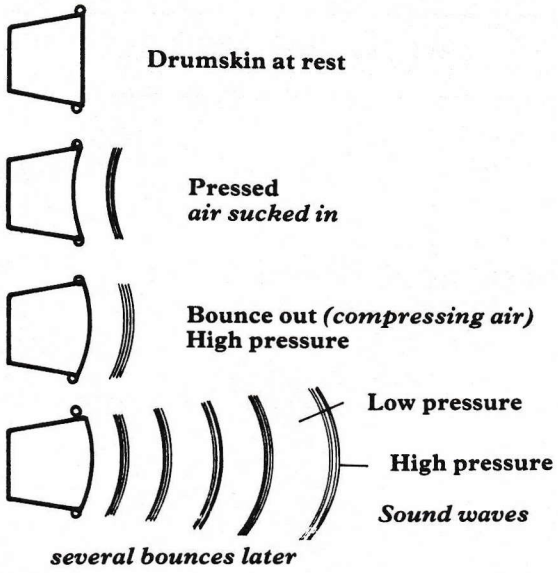


Figure 10.1 How a drum-skin produces sound by alternately compressing and decompressing air.

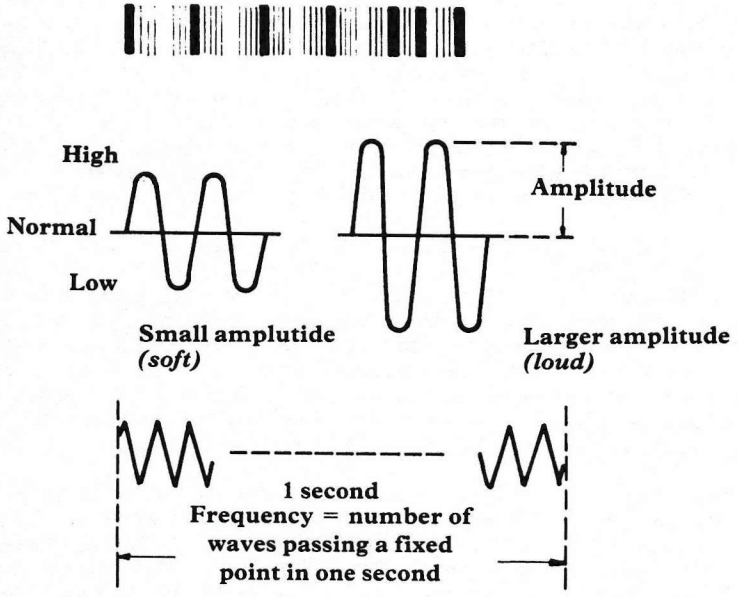


Figure 10.2 Sound waveforms, showing how the air pressure changes with time. The amount of change determines how loud the sound will be, and the number of changes per second is called the frequency, or pitch of the sound.

The amount of pressure change determines what we call the loudness of a note. This is measured in terms of *amplitude*, which is the maximum change of pressure of the air from its normal value. For complete control over the generation of sound, we need to be able to specify the amplitude, frequency, shape of wave, and also the way that the amplitude of the note changes during the time when it sounds.

The Spectrum 128 computer has two sound instructions, BEEP, and PLAY. In addition, there is the sound that you hear when you press a key. This 'key-click' sound can be changed with a POKE command. If, for example, you type `POKE 23609,10` (ENTER), you will get a longer note each time you press a key. This also alters the time that it takes Spectrum to deal with a key, and if you use a large number, you can almost paralyze the machine! Try `POKE 23609,200` to see the effect!.

Of the two instructions, BEEP and PLAY, BEEP is a simple instruction, and the notes from it have fixed pitch and amplitude. As I mentioned earlier, this amplitude is controlled by the volume control of your TV receiver, so that you can have it as loud or as soft as the TV receiver permits. The PLAY instruction is a much more complicated one, though it needs only a string following it. What makes it more complicated is that a set of letters and numbers in the string are needed to set up the sound, and it's designed mainly to produce music and other sound effects. We'll keep PLAY until later, and concentrate on BEEP for the moment.

BEEP has to be followed by two numbers, separated from each other as usual by a comma. Of the two, the first number is a duration number, and its value can range from 0 to 10, with fractions allowed. This number gives the time for which a note plays, in seconds. The second number is a pitch number that decides exactly what note will be played. The pitch number can be of values from –60 to +69. The negative notes are low notes, and the positive numbers give the high notes. A value of zero gives the note called middle C which is right in the middle of the musical scale. We'll look at these notes, and the number for the BEEP pitch command, later.

You shall have music . . .

The BEEP instruction is very useful for its purpose, but the Spectrum 128 computer has a lot more in store for you. A lot of computers are not really suited to working with music, because they require all of the instructions to be in number form. If you read music, or can work with sheet music, this is the last thing that you want. The ideal method of programming music would be to work with the named notes of music – and this is what the Spectrum 128 computer can do. It might appear to be the obvious thing to do, but very few computers do it! If you have no experience of music, however, this may seem rather puzzling to you. How do we go about writing down music? For each note, we have to specify what the note is (its pitch), how loud it must be, and for how long it is to be played. In written music, this is done by using a type of chart for the pitch, and different shapes of markings (notes) for the duration. Loudness is indicated by using letters such as `f` (loud) and `p` (soft). More than one letter can be used, so that `fff` means very loud, and `ppp` means very soft. Each sound is indicated by a note, a shape on the chart, and the shape of the note gives some information about the duration of the note. In addition to this, each piece of music will start with some advice about the speed at which the notes are to be played. One of these methods is a metronome reading. The metronome is a gadget which ticks at regular intervals, and the metronome reading for a piece of music is the number of metronome ticks per minute. A more ancient way of indicating speed is the use of Italian words like *allegro* (fast), *lento* (slow) and so on. What these speed settings decide is how many unit notes will be played in a minute. The unit note is called the *crochet*, so if a piece of music is marked at a metronome speed of 60 (pretty slow), then there will be 60 crochets played per minute. The durations of all the other notes are decided in comparison to this unit, the crochet. A *minim* sounds for twice as long as the crochet, a *semibreve* for twice as long as a minim, which is four times as long as the time of a crochet. The *quaver* sounds for only half the time of the crochet. A *semiquaver* sounds for only half the time of a quaver, which is quarter of the time of a crochet. The crochets and other timed notes are indicated by the shapes of the written notes, as Figure 10.3 shows, along with the control numbers that Spectrum uses for PLAY. In addition, symbols are used to indicate silences in the music, and these are based on the same idea of a unit duration of silence, and others which are twice, four times, half, or quarter. These silence marks are shown in Figure 10.4

Symbol	*Time*	*Name*	*Number*
𝅘𝅥𝅯	1/4	**Semiquaver**	1
𝅘𝅥𝅯.	3/8	**Dotted semiquaver**	2
𝅘𝅥𝅮	1/2	**Quaver**	3
𝅘𝅥𝅮.	3/4	**Dotted quaver**	4
♩	1	**Crochet**	5
♩.	1 1/2	**Dotted crochet**	6
𝅗𝅥	2	**Minim**	7
𝅗𝅥.	3	**Dotted minim**	8
𝅝	4	**Semibreve**	9

Figure 10.3 The symbols that are used in written music to indicate the time of each note, along with the Spectrum time numbers.

Symbol(s)	*Time*	*& number*
	1/4	&1
	3/8	&2
	1/2	&3
	3/4	&4
	1	&5
	1 1/2	&6
	2	&7
	3	&8
	4	&9

Figure 10.4 The symbols for silences in written music, with the Spectrum & numbers.

The pitch of a note is indicated in written music by placing it on to a kind of musical map which is called the *stave* (Figure 10.5). Piano music uses two of these staves, each consisting of five lines and four spaces. The upper stave is the treble stave, and it is used for writing the higher notes which will be played on the piano with your right-hand. The lower stave is the bass stave, the lower notes, played with the left-hand. Instruments which do not use a keyboard will normally have music written with only one staff. In addition to this representation of notes by position on staves, we also use the letters of the alphabet from A to G to name the notes.

The piano is the most familiar type of musical instrument, and its keyboard is set out so as to make it very easy to play one particular series of notes, called the 'scale of C Major'. The scale starts on a note that is called middle C, and ends on a note that is also called C, but which is the eighth note above middle C. A group of eight notes like this is called an 'octave', so that the note you end with in this scale is the C which is one octave above middle C. Because music (in the Western hemisphere, at least) is based on this group of eight notes, we use only the first seven letters of the alphabet in naming the notes. Why 7? Well, the eighth note is the end of one octave and the start of the next, so it bears the same name. The scientific basis of all this is that if you take middle C, and find the frequency of the sound of this note, then the C which is the next octave above middle C has precisely double the frequency value of middle C. The C below middle C has half the frequency of middle C, and so on. That's why the ancient Greeks always thought that music was a branch of mathematics.

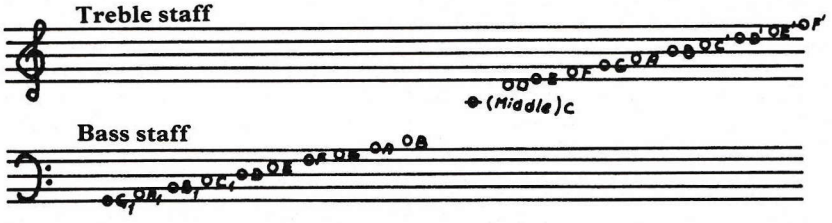


Figure 10.5 The staves of written music, with the names of the notes written in.

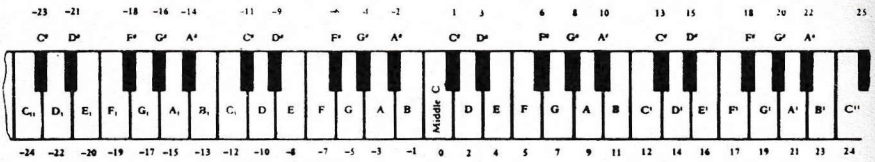


Figure 10.6 Part of the piano keyboard, showing Middle C. There is only one semitone between B and C, and between E and F. The BEEP numbers are also shown here.

The appearance of these keys on the piano keyboard, along with the BEEP numbers, is illustrated in Figure 10.6. Middle C is, logically enough, at the centre of the keyboard, and we move right for higher notes, left for lower notes. One of the complications of music, however, is that the frequencies of the notes of a scale are not evenly spaced out. The 'normal' full spacing is called a *tone* and the smaller spacing is called a *semitone*. Each scale will contain two semitones. On written music, middle C appears midway between the treble and bass staves.

The main instruction for playing music on your Spectrum 128 computer is the PLAY instruction which has to be followed by a string name. The string then contains all the information that is needed to produce the music. The notes are specified simply by their names, as used in music. These are the letters A to G, and we also use the signs # and $. The # sign means a semitone higher than the note indicated by the letter, so that #A is a semitone above A, the note a musician would call 'A-sharp'. Similarly, $A would mean a semitone below A, or 'A-flat'. For a Spectrum PLAY string, these signs are placed *before* the notes that they refer to, whereas on written music the sharp and flat signs are always placed following the note. In addition to the letter names of the notes, we can use other control letters to indicate the octave, volume, length, tempo and pauses. These letters *must* be typed in upper-case (capitals), and you will get an error message if you use lower-case. The octave letter is O, (not a zero!) and it has to be followed by a number whose range is 0 to 8. If you don't specify any 'O' value, the computer will set itself to O5. O0 means the lowest range of Spectrum computer notes; O8 gives the highest. This means that the Spectrum computer can play nine octaves of notes, which is more than the range of any ordinary musical instrument. In addition, the octave ranges overlap, so that you can cover two octaves of notes for each selected octave number. This is done by using the lower-case letters for the lower octave of notes, and the upper-case letters for the higher octave. The volume control letter, V, can be followed by a number whose range is 0 to 15. This lets us make music whose volume can change during the playing of the music. As you might expect, V0 gives the lowest volume, V15 the greatest. The computer sets itself to V15 if you don't specify anything different. We can, of course, still set the volume control of the TV to suit our own tastes. Unless you are using an amplifier, or have connected the Spectrum to a musical instrument by way of the MIDI port, you will get the best results from volume numbers in the range 10 to 15. A number used by itself controls the length of a note, and has to be in the range 1 to 12, though only 1 to 9 is really useful. This number is placed before the note number, and if there is a chance that this number might be written following another number like an octave number, you can use the letter N to separate the numbers. The pause or rest is a silent interval, and uses the ampersand sign &. It follows the same number scheme as note length, as Figure 10.4 indicates. If you don't

specify any other values, the computer uses length 5, the number for a crochet. The shortest note uses 1, the longest uses 9, and numbers 10 to 12 are used for special musical effects with bunches of notes.

```
10 LET a$="cdefgabC"
20 PLAY a$
```

Figure 10.7 A simple PLAY string and how to use it.

Time now for some illustrations. We'll start with Figure 10.7. This starts by defining a string a$. It consists of the notes that start at middle C. How do I know? Well, middle C on the Spectrum computer is the first note in octave number 5, so by starting with the letter c in the string, the first note that we get is middle C. The other notes have been written in sequence, but we need to remember that to get the higher 'C' as the last note, we have to use the letter C. If you use a small c, you'll get middle C again instead of the C above. The scale uses the default values of volume and speed (tempo).

```
  10 LET a$="T200V5cdefV12N7gabV
15N9C"
  20 PLAY a$
```

Figure 10.8 Altering volume as notes are played. Note the use of letter N to separate two numbers.

This is a simple scale, but it's a good piece of music to illustrate what can be done with this Spectrum command. Try Figure 10.8 now, to see what we can do with the volume V command, and the length number preceding the note. The first novelty in this program is T, which means 'tempo', and it controls the speed of playing the string. The setting of tempo is always equal to 120 crochets per minute unless you change it. The range of T is a curious one, 60 to 240, decided by the practical needs of music rather than by what the computer can do. The fastest tempo is obtained by using 240, the slowest by using 60 as you might expect. In the example, we have used the fast tempo of 200, but changed the volume and length of note settings. The reason for having separate tempo and length control letters and numbers is that you can get the tune sounding right by using the numbers to select the length of notes, and then use T right at the start to set whatever tempo you like, If you want to speed things up, use a high value for T, if you want a funeral march, use a low value. You can even write the string without a T, and then add it in later by a command like:

```
PLAY "T200"+A$
```

Envelopes

It's time now to look at some more revelations. What's the difference between a violin and a clarinet? Yes, I know, you'd look silly blowing a violin, but what's the difference in the sound? You can tell which of these instruments is playing a note, even if it's the same note at the same volume. The answer is that the envelope of the waves is different. The word 'envelope' needs some explanations. Envelope means the pattern of a note. A musical note does not consist of just one sound wave, but of

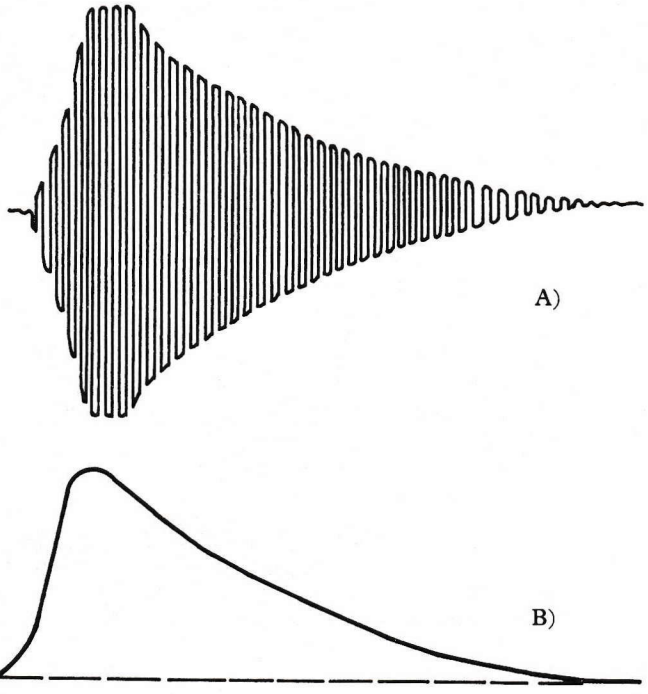


Figure 10.9 The envelope of a note: (**a**) how the amplitude of the waves change; and (**b**) the envelope outline that is the important part – only one half needs to be drawn.

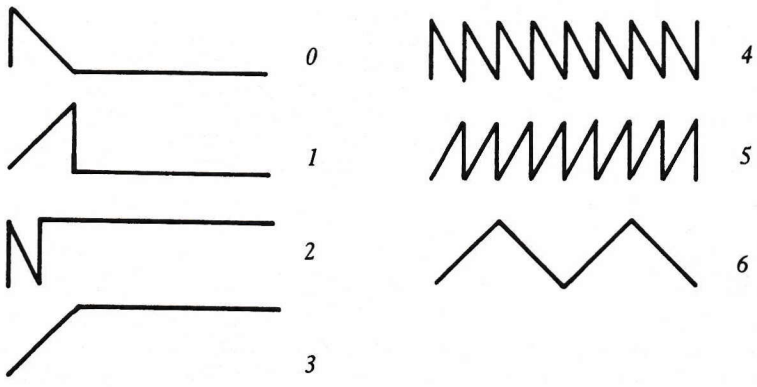


Figure 10.10 The standard envelope shapes of the Spectrum 128, and the code numbers that are used with them.

many. While a note is sounding, the volume need not be constant, though it is for the traditional 'electric-organ' type of note. For example, when you strike a piano key, the note starts very loud, and its volume then dies away as the string vibration dies away. It's envelope is therefore like the shape that is shown in Figure 10.9 – rising very sharply, then fading away. Each musical instrument produces its own type of envelope, and for some instruments, the effort of the player can alter the shape of the envelope. It's never easy to design your own shapes of envelopes with a computer, so the Spectrum computer allows you a choice of standard shapes. These are illustrated in Figure 10.10, along with the code numbers that are used to produce them. These code numbers are put into a program using the letter W.

Used on its own, W doesn't seem to do much, but when it is combined with U, it really becomes interesting. *W*, you see, over-rides all the V instructions, and the combination of W and U controls the volume of each note. U is simply used as a switch to turn on the effect of envelopes, and W can take values of 0 to 7, the code numbers for the eight different envelope shapes that you can use. There's another number, X that you sometimes need, but the Spectrum will normally set this value for you to give the best results. Figure 10.11 illustrates the effect that these two have when used together. The screen prints up the values as you hear the sounds. You will find that some of these notes are very useful, particularly the W7 one, which is a nice 'plucked string with tremolo' type of note. The effects of the W values are modified to some extent by the number that follows X. In the example, the same number has been used for all of the notes, but in general, it's better to have larger X values for the notes with the simple non-repeating patterns, like 0 and 3. The X number is, in fact, the number that controls the time that is needed for each *change* in an envelope. By making this number large for the envelope shapes that contain just one or two steps, we make it easier to hear the changes. For the repetitive patterns, however, we need to make the X values shorter so that the notes don't last for too long. In particular, if you use very large values of X, you will not hear the effect of the 'repeater' notes that you get with S values of 4 to 7 inclusive. The sound of musical notes is very much a matter of sound-it-and-see, and you always have to experiment to get exactly what you want. Notice, incidentally, how the number values for the W codes have been put into the PLAY string in Figure 10.11, by converting the number variable j into a string variable k$ and adding this into the end of the string that ends with a W.

Figure 10.12 shows an example of a tune written using the PLAY instruction. Points to watch for here are the use of a separate string for the envelope letters U, V, X and the use of the normal length numbers for the notes. The envelope that has been chosen here is the code zero one, which gives a good string note for this short phrase. You can now try the effect of other X values and other W values with this music, because all you have to alter is the first string, a$. Some experience of how these

```
10 LET a$="UW"
20 LET b$="X500Ncdefgabc"
30 FOR j=0 TO 7
40 LET k$=STR$ (j)
50 PRINT "Envelope No. ";j
60 PLAY a$+k$+b$
70 NEXT j
```

Figure 10.11 A program that allows you to hear the effects of the different envelopes.

```
10 LET a$="UW0X1000"
20 LET b$="5E3FE5DGC3DE5EDF3GF
5E3DC6D3C7C"
30 PLAY a$+b$
```

Figure 10.12 A simple tune using the PLAY instruction with envelopes.

effect sound is a lot more useful than any amount of explanations and advice. You'll find that only the W numbers 0 and 1 sound good with a large X number, and the repetitive patterns 0 to 7 need very short X numbers, in the region of 20 to 30. The envelopes 2 and 3 sound very faint unless suitable X values are used. This is because of the fairly fast tempo of the piece. Try the effect of omitting the X and X-number completely. You will need to put in an N at the end of a$ to do this, because otherwise the 2 at the end of a$ comes against the 5 at the start of b$, causing an error message. Without the X in place, the computer adjusts things so that you get a good loud note each time, though the effect may not be exactly what you want. Being able to do all this with the Spectrum computer makes it all the easier to transfer written music to the form of PLAY strings. The easiest way of getting music is to use the ordinary 'organ' note to start with. You can then add the envelope commands at the start of the string.

Sweet harmony

One of the many remarkable features of the Spectrum computer is that you can apply PLAY strings to three notes at a time, using three separate channels of sound. Three channels means that you can play up to three notes at once, and this feature allows you to have harmony. Needless to say, it requires a lot more thought, and you have to be careful to keep the three channels in step, or else what you get will certainly not be harmony. You might, of course, win a modern-music prize.

So far, we have used the variable names a$ and b$ at random, simply as names for strings that we wanted to play. The names a$, b$ and c$ have rather more significance, however, because they apply to three separate channels of sound. You can have entirely different notes playing in these three channels, so that if you have three separate music strings set up, you can use a statement such as PLAY a$,b$,c$ to get them all going together. This is the basis of harmony.

Figure 10.13 shows a simple example of two part harmony in action. The two strings have been written from a music score so that the notes will remain in time with each other, and the PLAY instruction uses both strings, separated by a comma. Note the

```
10 LET a$="O5N5E3FE5DGC3DE5EDF
3GF5E3DC6D3C7C"
20 LET b$="O4N7Cbag5bgCfgO3g7C
"
30 PLAY a$,b$
```

Figure 10.13 A piece of two-part harmony, using two channels of the three that are available.

difference – if you use `PLAY a$+b$` then the string will be played one after the other, but using `PLAY a$,b$` will play them together. According to the manual, you need to use the channel-selecting number in an `M` command, but all three channels seem to be turned on for music by default, and you need the `M` commands only to turn channels *off*. We'll look at the applications of `M` to noise effects later. If you want to play all three channels, you simply need to add another comma and another `c$` string of notes. For really good effect, put a `T6Ø` at the start of each string of notes (with `N` separating the `5E` that starts `b$`), record the tune on a cassette, and then play it back through a QUAD 100 watt amplifier into a pair of QUAD electrostatic loudspeakers . . . but don't blame me if all the ornaments fall off the walls!

What is a lot less simple is writing music for this extended PLAY routine. You should, unless you have some skills in composing, work from sheet music. Music for violin and piano, or soprano voice and piano, is particularly suitable. The instructions for Spectrum sound are, of all the computers I know, best adapted for musical transcription, and they include many thoughtful touches that indicate the presence of a keen musician in the team that wrote the BASIC. In a book of this size, it would be impossible to illustrate in detail all the capabilities of this excellent sound system, but a few are particularly well worth mentioning if you are interested in writing or transcribing music for the Spectrum. One is the use of brackets to enclose a phrase which has to be repeated. Using a pair of brackets means that all the notes within the brackets will be repeated once. If you place just one bracket at the end of a string, this will make the whole string repeat. Using two closing brackets will cause the repeat to go on indefinitely until the letter `H` appears in another string that is being played. The other effects that are very much aimed at musicians are the ability to use note-length numbers 10 to 12 for triplets – groups of three notes that are played in the time allocated for two. You can also make use of tied notes, by using something like `7_5C`, meaning that this C is to be played for the combined time of a minim and a crochet, but that the notes following will all be crochets unless otherwise written with different time numbers. Finally, you can put a comment into a string by enclosing it with exclamation marks such as `!melody!` without interfering with the way that the string of notes will play. Nice one!

Melody Channels

Select No.	*channel(s) selected*
1	A
2	B
3	A and B
4	C
5	A and C
6	B and C
7	A and B and C

Noise channels.

Select No.	*channel(s) selected*
8	A
16	B
24	A and B
32	C
40	A and C
48	B and C
56	A and B and C

These numbers can be added to the music channel numbers above.

***Note:* if no M numbers are used, all channels are turned *on*. Selecting channels with M will automatically turn all other channels off.**

Figure 10.14 The M code numbers for turning channels on and off. If you don't use M, then all channels are on.

Sounds unlimited

PLAY is the Spectrum computer's gift to the musician computer owner, now let's look at what Spectrum can offer to games enthusiasts who want sound effects. A sound effect is a variety of noise, something that can't be written into a musical score so easily as a tune. The Spectrum computer can use the PLAY instruction to allow you a range of effects which go far beyond the boundaries of written music and ordinary instruments. The instructions in the manuals don't exactly help you with this difficult type of effect, so I have dealt with it in a little more detail here. For a really comprehensive treatment, though, you would need a complete book devoted to such effects.

The use of PLAY for sound effects relies on the use of noise. Noise is a mixture of frequencies, unlike a musical note which always has one clear 'fundamental' frequency. Noise may nevertheless have a 'predominant' frequency, meaning that most of the noise frequencies are centered around this frequency instead of being spread around evenly over all the range of frequencies. The noise generating system of the Spectrum computer is most effective when you use Channel `A`, so that any string of this type has to be written as `a$`, and we have to make sure that the channels are not used for anything else. Figure 10.14 shows the code numbers that are used

with the control letter M to turn channels on and off. From the list, you can see that if we want only noise effects in Channel A, we need to start with M8 in the string. We can use the U, W, X instruction letters with their numbers as before, because most of the interesting noises depend on the use of envelopes. The octave numbers and letters don't mean much, but you must have a pitch letter in the string, even though altering it has no effect. The greatest effect is obtained by using the X number.

```
10 LET a$="M8UW0X2000(c))"
20 PLAY a$
```

Figure 10.15 A hammering sound effect.

Figure 10.15 illustrates a simple 'hammering' sound effect, using W0X2000. The sound is made repetitive by using the letter c enclosed in brackets, with two closing brackets. This makes the sound effect repeat indefinitely. You can stop it in this example by using the BREAK key, but in most cases when you want to use these effects this is not exactly completely satisfactory, because you don't usually want to press BREAK. We can get around this by using a second string that specifies for how long the noise string can play. Figure 10.16 illustrates this, using b$ for this second string. Note that you can't put PLAY a$+b$, because b$ can't play until a$ has finished, and it never finishes! A simpler method is to use as many pairs of brackets around the note letter as you need repetitions of that effect, and this is the method that is illustrated in the manual.

```
10 LET a$="M24UW0X2000(c))"
20 LET b$="&&&&&&&&H"
30 PLAY a$,b$
```

Figure 10.16 Determining the duration of a sound-effect string with another string of silences.

Now the use of noise really opens up sound in a way that will be a complete revelation if you have programmed the old 48K Spectrum. There are so many variations on X number and envelope number that even the simplest noise commands can produce some very exotic effects. Try the program in Figure 10.17 to hear what variations in the waveform number can give you, and for a lovely assortment of shots and propellor or rotor-blade effects, try Figure 10.18, in which shorter X numbers have been used. The sky's the limit when you get to this stage!

```
  5 FOR j=0 TO 7
 10 LET a$="M24UW"+STR$ (j)+"X2
000(c))"
 20 LET b$="&&&&&&&&H"
 30 PLAY a$,b$
 40 NEXT j
```

Figure 10.17 The effect of varying the waveform numbers.

```
  5 FOR j=4 TO 7
 10 LET a$="M24UW"+STR$ (j)+"X1
00(c))"
 20 LET b$="&&&&&&&&H"
 30 PLAY a$,b$
 40 NEXT j
```

Figure 10.18 An assortment of effects obtained by using shorter DSx numbers.

Chapter 11

Printers

Whenever your use of a computer extends beyond playing games that other people have written, there are two additions to your computer equipment that you will urgently want. One of these is a disk system, and that's a topic that is just too big for this book. The next must be a printer. In many cases, the printer has an even higher priority than the use of the disk system. The reasons for using a printer are obvious if you use the machine for even the smallest business purposes. You can hardly expect your accountants or your income-tax inspector to look at accounts that can be shown only on the screen. It would be a total waste of time if you kept your stock records with a computer, and then had to write down each change on a piece of paper, copying everything from the display on the screen. For all of these purposes, and particularly for word processing, the printer is an essential part of the computer system. Output on paper is referred to as 'hard copy', and this hard copy is essential if the computer is be of any use in business applications. For word-processing uses, it's not enough just to have a printer, you need a printer with a high-quality output with characters as clear as those of a first-class electric typewriter.

Even if your computer is never used for any kind of business purpose, however, you can run up against the need for a printer. If you use, modify or write programs, the printer can pay for itself in terms of your time. Trying to trace what a program does from a listing that you can see only a few lines at a time on the screen is totally frustrating. Quite apart from anything else, the BASIC of the Spectrum relies a lot on the use of GO TO for loops, and you might have to list a dozen different pieces of a program just to find where one GO TO might lead you to. The problem is even worse if you write your own programs. Even a very modest program may need a hundred lines of BASIC – and remember that the Spectrum 128 permits the use of very long lines. Trying to check a program of a hundred lines when you may be able to see only a dozen or so at a time is like bailing out a leaky boat with a teaspoon. With a printer attached to your Spectrum 128 you can print out the whole listing, and then examine it at your leisure. If you design your programs the way you ought to, using a 'core' and subroutines, then you can print each subroutine on a separate piece of paper. In this way, you can keep a note of each different subroutine, with variable names noted. On each sheet you can write what the subroutine does, what quantities are represented by the variable names, and how it is used. If you use the

MERGE action of the Spectrum, then you can construct programs painlessly using your library of tested subroutines.

Printer types

Granted, then, that the use of a printer is a high priority for the really serious computer user, what sort of printers are available? The answer is any type that comes with a 'serial interface', or for which a serial interface can be bought. Regrettably, this usually means spending rather more than the owners of other computers which use parallel interfaces. This apart, you have the usual choice of the different printer mechanisms. Printers that are used with small computers will use one of the mechanisms that are listed in Figure 11.1. Of these, the impact dot-matrix type is the most common. A dot matrix printer creates each character out of a set of dots, and when you look at the print closely, you can see the dot structure. Most of the dot-matrix printers are impact types. This means what is says, that the paper is marked by the impact of a needle on an inked ribbon which hits the paper. There are also thermal and electrostatic dot-matrix printers. These use needles, but the needles do not move. Instead the needles are used to affect a special type of paper. In the electrostatic printer (such as the old ZX printer), the needles are used to pass sparks to the paper, removing a thin coating of metal from the black backing paper. The thermal type of printer uses hot needles to make marks on heat-sensitive paper. Both of these printers require expensive special paper, and are unsuitable for serious business purposes, so we won't spend any time on them here. If you want a cheap printer for listings, there are better methods.

The ultimate in print quality at the moment is provided by the daisy-wheel printer. This uses a typewriter approach, with the letters and other characters placed on stalks round a wheel. The principle is that the wheel spins to get the letter that you want at the top, and then a small hammer hits the back of the letter, pressing it against the ribbon and on to the paper. Because this is exactly the same way as a typewriter produces text, the quality of print is very high. It's also possible now to buy a combination of typewriter and daisy-wheel printer. This looks like a typewriter, with a normal typewriter keyboard, but has an interface connection for a computer. You can use it as a typewriter, and then connect it to the computer and use it as a printer. Machines of this sort are made by leading typewriter manufacturers such as Silver Reed, Brother, Triumph-Adler, Smith-Corona, and others. If you need a typewriter as well as a printer, then this type of machine is an obvious choice.

The third kind of mechanism of interest is the graphics printer. This is a remarkable mechanism which uses four miniature ball pens to mark the paper direct, with no ribbon. It can be used for graphics work, and when it is used as a printer, the letters are drawn rather than printed. Because four pens are used, the markings can be in four different colours. Printers of this type are not expensive (as printers go) and can be very useful, particularly if you want graphics output in colour. Another type of printer that is now becoming available is the ink-jet printer, which operates by shooting fine jets of ink at the paper. This one shares the disadvantage of the thermal and the electrostatic types that you get only one copy. Impact printers have the great advantage that you can get an extra copy by using a sheet of carbon paper and another sheet of plain paper. You can also buy listing paper which has a built-in carbon, or which uses the NCR (No Carbon Required) principle to produce two copies.

Dot Matrix

impact
thermal
electrostatic

Type Impact

type stalk
daisywheel
thimble
type-band

Plotters

graphics printers
X-Y plotters

Ink jet

single colour
multi-colour

Laser

laser fast printer

Figure 11.1 A list of printer mechanisms.

Interfaces

The printer has to be connected by a cable to the computer, so that signals can be passed in each direction. The computer will pass to the printer the signals that make the printer produce characters on the paper, but the printer must also be able to pass signals to the computer. This is because the printer operates much more slowly than the computer. Unless the printer contains a large memory 'buffer', so that it can store all the signals from the computer and then get to work on them at its own pace, some sort of *handshaking* is needed. This means that the printer will accept as many signals as its memory will take, and then sends out a signal to the computer which makes the computer hang up. When the printer has completed a number of characters, (one line, one thousand, or possibly just one character), it changes the handshake signal, and the computer sends another batch. This continues until all of the text has been printed. This can mean that you don't have the use of the computer until the printer has finished. Printers can be very slow, particularly daisy-wheel and plotter types. Even the fastest dot-matrix printers can make you wait for a minute or more for a listing. The Spectrum uses a built-in buffer with its serial interface, which means that you can usually get on with other things while the printer is busy. This is a considerable advantage, and it makes the extra cost of a serial printer easier to bear.

The serial interface sends the signals out one bit at a time. This means that at least seven signals have to be sent for each character, and in practice this number must be ten or eleven, to allow for 'start and stop' signals which are used to mark where the signals for each character start and stop. This system uses less cabling, because only two strands need to be used for signals, and the cables can be longer, because there's no risk of one signal interfering with another. The standard system is called 'RS-232', and you can easily buy a cable that will connect your Spectrum to any RS-232 printer, or to any other device that uses the RS-232 connection. You must then arrange for printing to take place by a FORMAT command. This will take the form `FORMAT"P";rate`, where `rate` means the number of bits of signal that will be transmitted per second. For a lot of printers this can be set to fast rates, and if you can set your printer to a rate (the baud rate) of 4800, then using `FORMAT|"P";48ØØ` will make the Spectrum match the printer. If the printer has settings for start bits, parity, data bits, and stop bits, set it to one start bit, eight data bits, no parity, and two stop bits, since this matches the way that Spectrum is arranged.

A problem that you are bound to run up against when you use any printer is that of line feed and carriage return. A lot of computers send out only one code number, the carriage return code (13) at the end of a line. Other machines send both the line feed (code 10) and carriage return codes. Printers are arranged, therefore, so that either possibility can be catered for by a switch. If you connect your printer and find that everything is printed on one line, then don't return the printer. Just look in the manual for the printer, and find the switch that alters the line-feed setting. If, on the other hand, you find that each line is double-spaced, then this switch will have to be set to the opposite position.

The EPSON RX-80

The EPSON range of printers has for a long time been the most popular range of moderately-priced printers, offering good print quality at reasonable prices. The RX-80 is one of this line, and all the listings in this book were printed on this machine. If, however, you are offered a second-hand MX-80, then this also is a good buy. A particular feature of the EPSON range is that the print heads can easily be replaced when they wear out. My old EPSON MX-80 is just beginning to show signs of head wear after printing half-a-million words, so it might not be a problem for you!

The standard version of the RX-80 uses pin-feed, but the RX-80F/T can take any form or paper, including rolls. You have to pay extra for a paper roll holder, but if you are handy with wood, this is something that you could easily make for yourself. The advantage of using the F/T version is that plain non-perforated paper rolls are very much cheaper to buy, and it also means that you can use plain paper sheets if you want to. When you use a lot of paper for listings, this can be a great saving. Paper width of 4" to 10" in pin-feed can be used, so you can buy whatever paper size is on offer. If you use the F/T option, you can then buy the teletype rolls, which are eight and a half inches wide.

The RX-80 can be obtained with a serial interface as an extra, plugging inside the case, and switchable so that you can also use a parallel (Centronics) interface if you want without taking the serial interface out. The printer itself offers a full set of upper or lower-case letters, and you don't have to go through any elaborate antics to select which one you want. This is because most of the selections are made by sending control codes to the printer. These codes all start with CHR$(27) – and that's a snag. The reason is that the print routines of the Spectrum will not pass on

CHR$(27) to a printer!. You can get around this with a little bit of software cunning. If you follow the FORMAT command with a POKE 23349,39:POKE 2335Ø,1 (ENTER) command, then you can send any code you like to the printer. If you put the FORMAT and POKE lines into a short program, followed by whatever codes you might want to send to the printer, then by POKE 23349,36, the codes will be accepted each time the program runs, and the printer channel of the Spectrum will be returned to normal, accepting no more control codes. If, on the other hand, you don't use the POKE 23349,36 command again, the CHR$(27) commands can be used in programs, so that you can print normal, condensed, emphasised, double width, and all of the other varieties, under program control. This makes it very easy to produce good headings, produce words in bold type or italics, and to underline. For a lot of word-processing actions, the RX-80, or its successor, the LX-80, can be a very satisfactory low-cost alternative to a daisy-wheel. International character sets (USA, France, Germany, England, Denmark, Sweden, Italy, Spain, Japan, Norway) can be printed, and are under software control. This means that selection is done by printing CHR$ numbers rather than by altering switches on the printer itself.

The CGP-115 4-colour graphics printer

One of the most popular small graphics printer mechanisms is made under the trademark of ALPS. It's Japanese, and in place of the mechanisms that are used by most printers, it actually *draws* its characters with a set of four miniature ball pens. The reason for the set of four is that this allows printing in four different colours – black, blue, red and green. The mechanism is made into boxed units by many manufacturers, and sold under a wide variety of names, but it is most easily obtained from Tandy stores, under the Tandy code number of CGP-115. This version includes both a Centronics and a serial interface, which makes the printer useable on practically any microcomputer which uses reasonably standard interfaces. Since the Tandy stores offer a good service on spares (pens, paper etc) and trouble-shooting, it makes sense to buy the Tandy version as there is a Tandy store in most large towns. In addition to being used as a printer, however, this machine acts as a graphics plotter, and you can draw diagrams and other pictures by means of instructions sent from the computer.

The printer uses a plain paper roll which is 4.5 inches wide. Tandy stores sell three rolls, each about 145 to 150 feet long, for just under £5. These paper rolls are also used by a wide variety of adding machines, so if you haunt your local office supply

stores, you may find alternative sources at lower prices. The paper is tightly gripped by the printer, because it is moved around a lot in the course of printing. The printing carriage consists of a holder which is loaded with four miniature ball pens. This holder can be rotated so that one pen is touching the paper. Printing is done by moving the pen holder from side to side, and the paper up and down, and is such a fascinating sight that you'll probably print listings over and over again just for the pleasure of watching the mechanism! I know that I did. When the printer is switched on, it goes into a 'pen-test' routine, slowly drawing a square in each colour so that you can check that none of the pens has run dry. They have a surprisingly long life, and each pack of three pens costs around £1.99 from Tandy stores. You won't find alternative supplies quite so easily in this case!

Normally, the CGP-115 acts as a printer, and you can use it to print listings. It is nothing like so fast as the EPSON printer, but the results are easy to read. The enormous advantage of using the Tandy printer, however, is that it can be used as a graphics plotter. This means that if you send suitable instructions to the printer, it will draw diagrams. The instructions are not the same as the graphics instructions of the Spectrum 128 (or any other computer), but this is not a disadvantage. If at some stage you change to another computer, the Tandy printer will still be useful, and the graphics programs that you have used with Spectrum 128 can easily be adapted to another computer. This is very useful to know if your household is on the verge of becoming a two-computer family!

The CGP-115 has a small set of four switches at the back which can be used for setting up the printer. For the Spectrum 128, you need to set the switches for serial interface, line feed with carriage return, and normal size print. The baud rate for the printer is 600, so you need to have the command `FORMAT"P";600` issued before you can use the printer. You will need a special cable for this printer, because it does not use the standard 25-pin 'D' plug that other serial printers use. Nevertheless, it's an excellent buy, and a very useful accessory for your Spectrum.

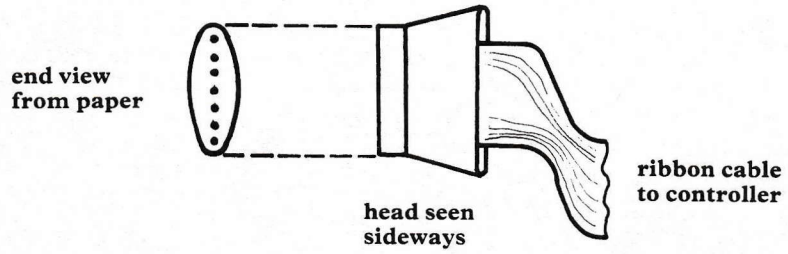


Figure 11.2 The principle of the dot-matrix printer.

Appendix A

Cassette loading problems

All cassette recorders work in the same way, by pulling the tape from one reel of the cassette past a miniature electromagnet that is called the tape head, and then winding the tape on to the other reel. The tape has to make very close contact with this tape head, and this is ensured by using a felt pad on a spring pressing against the tape. This 'pressure pad' is part of the cassette, and when you put a cassette into the recorder and press the PLAY switch, the tape head moves up to the tape, sandwiching it between the tape head and the pressure pad.

One very common source of trouble will be caused if the tape cannot be pressed tightly enough against the head. This will happen if there is a film of dirt on the head or on the tape, and this is why you have to take some care of your tapes. You need to take much more care with your tape head. The important thing is that you must NEVER touch a tape head with your fingers, anything metal, or even hard plastic. If your tape head has become dirty, then buy one of the many kits of cleaning materials that you can get from Curry's or Boots, and follow the instructions. You can get 'cleaning tapes' also, but I have never found them to be very effective. Clean the head once a month if you make a lot of use of the recorder.

There is another less obvious source of problems with cassette recorders. The important part of the tape head is a tiny slit in the metal, so small that you need a microscope to see it clearly. When a recorder is manufactured, the tape head should be positioned so that this slit will be at 90 degrees to the tape. The positioning is done by clamping one side of the tape head, and leaving the other one adjustable by means of a screw which will tilt the head as it is turned. This adjusting screw should have been set before the cassette recorder left the factory, but it seldom is. If the angle of the slit on your tape head is not exactly identical to the angle on the recorder that created your tapes, then good loading is almost impossible. You can, however, check the angle of your tape head for yourself, using a tape that you find difficult to load. This is possible only if your recorder allows the adjustment to be operated from outside, though. The tell-tale sign of this is a small hole in the casing just in front of where the cassette is placed. If your recorder has this adjusting hole, then all you need to get better loading is a small watchmaker's screwdriver.

First of all, insert a cassette with a long program that is difficult to load. Remove the cables between the recorder and the computer, so that you can hear the sound of the cassette playing through the loudspeaker of the cassette recorder. Press the PLAY key, and set the volume control to a comfortable listening level. Put any tone control to the maximum treble position, so that the sound is sharp and clear, not dull and muffled.

Now insert a watch-maker's screwdriver into the hole for head adjustment, and locate the blade in the head of the adjusting screw. This can be reached only when the tape is playing, because the heads are not in place at other times. On some recorders, the adjustment screw is difficult to reach, because it is under the flap, and you have to operate the recorder with the cassette in place but the flap open. If there is no hole for the screwdriver you can't adjust it! The adjustment consists of rotating the screw by about half a turn in one direction and listening to the sound. If the change causes the sound to change to a shriller, brighter note, keep adjusting in the same direction until it becomes dull again, then turn back until you get to the position at which the sound is brightest. If you find that your first adjustment makes the sound duller, then turn the screwdriver in the opposite direction. Once the adjusting screw is in the position for the sharpest clearest sound, you will find that conditions are ideal for loading. Stop the playing, rewind the tape, and re-connect to your Spectrum. You should find that you can now load the tape perfectly, providing that it was not made with the EAR plug in place.

One snag with this adjustment is that if you have a set of tapes that you made with the same recorder before adjustment, they will probably not load correctly afterwards. Another problem is that you often find variations between commercial tapes, so that you sometimes have to adjust the head to load a new tape that you have bought. If you have to do this, it's a good idea to get back to the original setting afterwards, otherwise you can end up with tapes made at all sorts of settings. This is just one of the things that drives the serious computer owner to disks sooner or later!

Appendix B

Editing

One of the greatest changes between the new Spectrum 128 and the older Spectrum models is the provision of a really good editing system on the new machine. Editing means correcting mistakes and rearranging programs, and on some computers it can be a very tedious and awkward business. The Spectrum 128 has a system called *screen editing*, the best type there is, and some time spent in mastering it will make your computing much more enjoyable. In addition, the Spectrum 128 has some editing actions that are obtained by pressing the EDIT key on the left-hand side of the keyboard. Not all of the actions are useful to you when you start programming, and we'll begin by looking at the straightforward editing actions on the screen.

The two main times when you want to change something are when you have just typed a mistake, and when you are looking at a listing that refuses to run correctly. When you are typing a program, and you have just made a mistake, then if you have not pressed ENTER, the easiest way of correcting is to use the DELETE key to wipe out characters to the left of the cursor. This can be a bit of a waste if the mistake is at the start of a line, and you have reached the end, so you can use the left-arrow key to move the cursor to where the mistake is. If you want to insert a letter (you typed PINT instead of PRINT, perhaps, because of a raging thirst), then place the cursor over the letter that follows the missing one, the I in this example. Now type the missing letter. Obviously, the same applies if you have two or more missing letters, or if you want to add words. When you press the ENTER key, the change will be made, or you can use the right-arrow key to get back to another part of the line, or to the end, for more typing before you press ENTER. If you need to delete a character, then place the cursor over the *next* character, and press the DELETE key once.

Once you have pressed the ENTER key, you can't alter a mistake simply by pressing the DELETE key, but correction is still easy. All you have to do is to place the cursor by using the arrowed keys, and proceed as usual with the DELETE key or by typing new characters. Whatever you type on the screen will be put into the lines when you press ENTER. You can even change the number of a line in this way, but when you press ENTER you will have two similar lines, one with the old line number and one with the new line number. You will probably want to delete one of these lines by typing its line number and then pressing ENTER. If you have needed to insert a lot of

new lines with numbers that are not in the usual sequence of 10, 20, 30 and so on, you can renumber all of the lines. This is done by pressing the EDIT key, selecting the Renumber option, and then pressing ENTER. This will renumber all of your lines starting with 10 and going up in steps of ten. The numbers that follow GO TO and GO SUB will automatically be altered when a program is renumbered in this way, *but*, if you use the GO TO repeat type of loop, you will have to alter the numbers for yourself. The other EDIT options are less useful – the PRINT option does nothing unless you have a printer attached and correctly set up, and the SCREEN option allows automatic listing only on the small screen of two lines at the bottom of the main screen.

This automatic listing system can cause problems until you get used to it. The principle is that after a program has run or stopped, pressing ENTER causes a listing to appear again, ready for editing. The cursor appears between lines, so that if you type RUN or GO TO 50 or any other command it will be obeyed, not added to the listing. What you have to watch is the cursor position. If you move the cursor from a blank space to a line, using the arrowed keys, then you are ready to edit that line. Suppose, for example, you move the cursor from a blank space between lines 100 and 110 so that it sits over the first letter of line 110, which is `11Ø REM`. If you now type RUN, all that happens is that the line becomes `11Ø RUNREM`, which is probably not what you wanted! The automatic listing is very useful when you are testing a program, because you don't have to keep typing LIST, and you can get to any part of the listing by using the arrowed keys. If you use the SCREEN option of EDIT, then the listing appears only on the bottom lines, and you can see a listing on the main screen only by using LIST. After using SCREEN, you can't edit on the main screen, only on the small screen. You can get back to normal editing by selecting the `Exit` option, then 128 BASIC. The SCREEN option is particularly useful when you have a printer attached and you want to make a paper copy of something on the main screen. This can be done by using the command COPY, and when you use the small screen for a command like this, the command word doesn't appear on the paper copy.

If you have the optional editing keypad, which may be supplied as standard at some stage since there is space for it in the box, then several other editing commands become available. You can move the cursor by more than one character at a time for example, and delete more than one character at a time. These actions are useful, but not essential, and unless you need to edit text or very long programs, then you may very well find that you don't need the separate editing keypad very much.

Appendix C

The MIDI interface

The letters MIDI stand for Musical Instrument Digital Interface, and the MIDI system has been devised to allow electronic instruments, particularly synthesisers and drum effects, to be controlled by the computer. The MIDI system is an international standard, and a full explanation would require much more space than is available in this book. What follows, then, is just a very brief note about the system so that you know what is involved. If you are interested in electronic musical instruments then you may have met the MIDI system before, and we'll concentrate on the computer end of the system.

To start with, you need a suitable connecting cable which you can get from your local Spectrum dealer. This will have the Spectrum RS-232 plug at one end and the standard MIDI plug at the other, and you should have the connections made between the computer and the instrument before you try to use MIDI commands. With this link in place, the PLAY instruction can now control the musical instrument, not just the sound of the Spectrum itself. To do this, you need to use a few additional command letters in your music strings. Each string must now start with the letter Y, followed by a digit 1 to 16 for channel number. This allows you to control instruments with up to 8 channels (not 16), but if you use the same channel number for more than one string you can play up to eight notes on one instrument. Alternatively, if you have more than one instrument connected through the MIDI interface, you can allocate channel numbers so that you can control up to eight instruments! If you don't use a digit following the letter Y in a music string, then channel 1 will be used. If the Y is omitted, all sound will be sent to the Spectrum sound generator rather than to the MIDI interface.

You may also have to alter controls on the instrument that you use. Synthesisers in particular that operate on several channels are usually arranged so that all notes are played on Channel 1 as well as on whatever channel has been commanded by your Y number. There will be a control to alter this on some instruments (switch from OMNI to POLY, for example), or it can be changed by sending a control code number in the music string. This control code number will have to be obtained from the manual for the instrument, and will be sent following a letter Z in the music string. When you are using the Spectrum 128 with a synthesiser, you need use only

the simple music strings, because the envelope will be provided by the synthesiser. Working in this way, you can control the volume of most synthesisers by using the V numbers in the music string. A few synthesisers will state in their manuals that they do not respond to key velocity, and these types will ignore any V commands in the music strings. If your synthesiser gives a list of 'key velocity' numbers, then the number that is sent is eight times the value of the V number. In other words, if you use V6 then the number that is sent to the synthesiser is 48.

INDEX

Spectrum 128 Companion

(Spectrum 128 and Plus-2)

The Spectrum 128 is the ultimate Spectrum - low priced, very capable and easy to use. This book concentrates on the 128k mode of Spectrum BASIC which makes the machine so attractive to both new and experienced computer users alike. 128 mode BASIC holds many surprises in store for the seasoned Spectrum user; for example, it demands the full typing in of statements. With its attractive new keyboard and the excellent new sound capabilities, the Spectrum 128 is a firm market leader. The instructions for sound control are the best available, but they do require some understanding of music and of programming techniques if the user is to get the best from them. This book therefore concentrates on the 128's new abilities, but not forgetting compatibility with older products such as microdrives.

The book guides you through the BASIC programming language of the Spectrum 128 and Plus-2, using *short* examples that will not over-tax the one-finger typist. It also includes rather more than many older books on program design, the appearance of text on the screen, the presentation of data etc. It also includes much practical advice on such subjects as tuning your TV receiver to the best advantage, the benefits of using a TV/monitor, adjusting the azimuth of a cassette recorder for use with machines made prior to the Plus-2 etc. Start reading here to get more enjoyment from your Spectrum 128 or Plus-2!

GLENTOP

ISBN 1-85181-080-3

9 781851 810802

£5.95

GLENTOP PRESS LTD

Standfast House,
Bath Place,
High Street,
Barnet,
Herts EN5 5XE